VILNIUS GEDIMINAS TECHNICAL UNIVERSITY

Justas KAZANAVIČIUS

RESEARCH ON LEGACY MONOLITH APPLICATIONS DECOMPOSITION INTO MICROSERVICE ARCHITECTURE

DOCTORAL DISSERTATION

TECHNOLOGICAL SCIENCES
INFORMATICS ENGINEERING (T 007)

Vilnius, 2024

The doctoral dissertation was prepared at Vilnius Gediminas Technical University in 2019–2024.

The dissertation will be defended at the public meeting of the Dissertation Defence Council of the Scientific Field of Informatics Engineering in the SRA-I Hall of Vilnius Gediminas Technical University at **10 a.m. on 21 May 2024**.

Address: Saulėtekio al. 11, LT–10223 Vilnius, Lithuania.
Tel.: +370 5 274 4956; fax +370 5 270 0112; e-mail: doktor@vilniustech.lt

A notification on the intended defence of the dissertation was sent on 19 April 2024. A copy of the doctoral dissertation is available for review at the Vilnius Gediminas Technical University repository https://etalpykla.vilniustech.lt and the Library of Vilnius Gediminas Technical University (Saulėtekio al. 14, LT–10223 Vilnius, Lithuania) and the library of Kaunas University of Technology (K. Donelaičio st 20, LT-44239 Kaunas, Lithuania).

Vilnius Gediminas Technical University book No 2024-018-M

https://doi.org/10.20334/2024-018-M

VILNIAUS GEDIMINO TECHNIKOS UNIVERSITETAS

Justas KAZANAVIČIUS

MONOLITINĖS ARCHITEKTŪROS PROGRAMŲ MIGRACIJOS Į MIKROSERVISŲ ARCHITEKTŪRĄ TYRIMAS

DAKTARO DISERTACIJA

TECHNOLOGIJOS MOKSLAI,
INFORMATIKOS INŽINERIJA (T 007)

Vilnius, 2024

Abstract

Microservice architecture is becoming the *de facto* industry standard for building new enterprise applications. According to the International Data Corporation, almost 90% of North American enterprises already use microservice architecture to develop new and modernise legacy applications. Companies aiming to remain competitive have started modernising their legacy monolithic systems by decomposing them into microservices. However, extracting microservices from legacy monolithic software is an extremely complex task.

Although the topic of monolithic software migration into microservice architecture has already been explored by scientists and software engineers, it is a complex and relatively new challenge; therefore, little research exists on its many parts, such as database adaptation during the migration and communication establishment between microservices. Most research primarily focuses on microservice identification within monolith applications and source code decomposition into microservices. A new migration approach is proposed to bridge this gap. It consists of code decomposition and covers communication establishment and data management.

The dissertation consists of an introduction, four chapters, and general conclusions. The first chapter introduces microservice and monolithic architectures and discusses the existing migration from monolithic to microservice architecture methods. In addition, three main parts are identified, and deeper research is provided for code extraction methods, communication between microservices, and data management. It also provides evaluation of existing methodologies for monolith decomposition into microservices. The same enterprise application was decomposed into microservices using three different methods. Based on the proposed evaluation criteria, the advantages and disadvantages of each decomposition method were determined. The second chapter presents the proposed approach for migrating legacy monolithic applications into microservices. The third chapter presents experimental research on possible communication technologies. Five communication technologies, such as HTTP Rest, RabbitMQ, Kafka, gRPC, and GraphQL, have been evaluated and compared using the proposed evaluation criteria. The fourth chapter presents an experimental evaluation of the proposed approach of monolithic database migration into multi-model polyglot persistence.

The dissertation's results were published in 4 scientific publications, 2 of which were in reviewed scientific journals indexed in the *Clarivate Analytics Web of Science* database and presented at four international conferences.

Reziumė

Mikroservisų architektūra tapo *de facto* pramonės standartu kuriant naujas taikomąsias programas. Tarptautinės duomenų korporacijos duomenimis, beveik 90 % Šiaurės Amerikos įmonių jau naudoja mikroservisų architektūrą naujai programinėj įrangai kurti ir senai programinei įrangai modernizuoti. Siekdamos išlikti konkurencingos, įmonės pradėjo modernizuoti savo monolitines programas, išskaidydamos jas į mikroservisus. Tačiau mikroservisų išgavimas iš senos monolitinės programinės įrangos yra labai kompleksinė užduotis.

Nors monolitinės programinės įrangos perkėlimo į mikroservisų architektūrą tema nagrinėta mokslininkų ir programinės įrangos inžinierių, tačiau tai yra palyginti naujas ir kompleksinis iššūkis. Daugumos tyrimų pagrindinis dėmesys skiriamas mikroservisams identifikuoti monolitinės programos išeities kode. O tokios temos kaip ryšio tarp mikroservisų užmezgimas ir duomenų bazės adaptacija yra vis dar mažai tyrinėjamos. Siekiant užpildyti šią spragą, siūlomas naujas perkėlimo metodas. Jį sudaro ne tik išeities kodo išskaidymas, bet ir ryšio užmezgimas tarp mikroservisų bei duomenų bazės adaptacija.

Disertacija sudaryta iš įvado, keturių skyrių ir bendrųjų išvadų. Pirmajame skyriuje pristatomos mikroservisų ir monolitinės architektūros bei aptariami esami migracijos iš monolitinės architektūros prie mikroservisų architektūros metodai. Papildomai išskiriamos trys pagrindinės perkėlimo dalys ir atlikti išsamesni tyrimai: kodo išgavimo metodų, komunikacijos tarp mikroservisų ir duomenų bazių adaptacijos. Pirmajame skyriuje taip pat tiriamos esamos monolitinės programinės įrangos skaidymo į mikroservisus metodikos. Ta pati programa buvo išskaidyta į mikroservisus, taikant tris skirtingus metodus. Remiantis pasiūlytais vertinimo kriterijais, nustatyti kiekvieno migravimo metodo privalumai ir trūkumai. Antrajame skyriuje pateikiamas siūlomas migracijos iš monolitinės architektūros į mikroservisų architektūrą metodas. Trečiajame skyriuje pristatomi eksperimentiniai komunikacijos technologijų tyrimai. Penkios komunikacijos technologijos, tokios kaip HTTP Rest, RabbitMQ, Kafka, gRPC ir GraphQL, buvo įvertintos ir palygintos pagal siūlomus vertinimo kriterijus. Ketvirtajame skyriuje pateikiamas siūlomas perkėlimo metodas ir eksperimentinis monolitinės duomenų bazės transformacijos į daugiamodelį poliglotinį modelį įvertinimas.

Disertacijos rezultatai buvo publikuoti 4 mokslinėse publikacijose, iš kurių 2 publikacijos, publikuotos žurnaluose, indeksuojamuose *Clarivate Analytics Web of Science* duomenų bazėje, ir pristatyti 4 mokslinėse konferencijose.

Notations

Symbols

t – time used to process the message (liet. *laikas, naudojamas pranešimui apdoroti.);*
Mi – microservice with index I (liet. *mikroservisas su indeksu I.*);
Req. – request (liet. *užklausa.*);
Res. – response (liet. *atsakymas.*);
"→" – request/response operation (liet. *užklausos/atsakymo operacija.*);
RPS – requests per second (liet. *užklausos per sekundę.*);
1 – one relationship in the entity relationship diagram (liet. *vienas ryšys objekto santykių diagramoje.*);
N – many relationships in the entity relationship diagram (liet. *daug ryšių objekto santykių diagramoje.*);
[] – collection (liet. *sąrašas.*);
"+" – means that the criteria is an advantage (liet. *reiškia, kriterijai yra privalumas.*);
"-" – means that the criterion is a disadvantage (liet. *reiškia, kriterijus yra trūkumas.*);
κ – Fleiss' kappa inter-rater agreement (liet. *Fleisso kappa vertintojų susitarimo koeficientas.*).

Abbreviations

AMQP – advanced message queuing protocol (liet. *išplėstinis pranešimų eilės protokolas.*);

API – application programming interface (liet. *taikomųjų programų programavimo sąsaja.*);

AQL – ArrangoDB query language (liet. *ArrangoDB užklausos kalba.*);

CAP – consistency, availability, and partition tolerant (liet. *nuoseklumas, prieinamumas ir atsparumas skaidiniams.*);

CI/CD – continuous integration and continuous deployment (liet. *nuolatinis integravimas ir nuolatinis diegimas.*) ;

DDD – domain-driven development (liet. *domenu pagrįsta plėtra.*);

DevOps – development operations (liet. *plėtros operacijos.*);

DMBS – database management system (liet. *duomenų bazių valdymo sistema.*);

DNS – domain name system (liet. *domenų vardų sistema.*);

GRPC – Google remote procedure call (liet. *google nuotolinės procedūros skambutis.*);

HTTP – hypertext transfer protocol (liet. *hiperteksto perdavimo protokolas.*);

HTTPS – hypertext transfer protocol secure (liet. *saugus hiperteksto perdavimo protokolas.*);

ID – identification (liet. *identifikavimas.*);

IDE – integrated development environment (liet. *integruota plėtros aplinka.*);

IP – Internet protocol address (liet. *interneto protokolo adresas.*);

IT – information technology (liet. *informacinės technologijos.*);

JSON – JavaScript object notation (liet. *JavaScript objekto žymėjimas.*);

KLOC – thousands of lines of code (liet. *tūkstančiai kodo eilučių.*);

OS – operating system (liet. *operacinė sistema.*);

RAM – random access memory (liet. *laisvosios kreipties atmintis.*);

REST – representational state transfer (liet. *reprezentacinis būsenos perdavimas.*);

RPC – remote procedure call (liet. *nuotolinės procedūros skambutis.*);

SOA – service-oriented architecture (liet. *į servisus orientuota architektūra.*);

SQL – structured query language (liet. *struktūrinės užklausos kalba.*);

SSD – solid-state drive (liet. *kietojo kūno diskas.*);

SSI – standard settlement instruction (liet. *standartinė atsiskaitymo instrukcija.*);

VM – virtual machine (liet. *virtuali mašina.*);

XML – extensible markup language (liet. *išplečiama žymėjimo kalba.*);

Definitions

ACID – acronym refers to the four key properties of a transaction: atomicity, consistency, isolation, and durability (liet. *akronimas reiškia keturias pagrindines transakcijos savybes: atomiškumą, nuoseklumą, izoliaciją ir ilgaamžiškumą.*).

AVAILABILITY ZONE – in the context of cloud computing, an availability zone is a public cloud provider's data centre that contains its own power and network connectivity (liet. *debesų kompiuterijos kontekste pasiekiamumo zona yra viešasis debesies paslaugų teikėjo duomenų centras, kuriame yra atskira galia ir tinklo ryšys.*).

BASE – acronym refers to the three key properties of consistency: available, soft state, and eventually consistent (liet. *akronimas reiškia tris pagrindines nuoseklumo savybes: prieinama, minkšta būsena ir galiausiai nuosekli.*).

DOCKER – container image, which is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings (liet. *atskiras vykdomasis programinės įrangos paketas, kuriame yra viskas, ko reikia programai paleisti: kodas, vykdymo laikas, sistemos įrankiai, sistemos bibliotekos ir nustatymai.*).

OPENSHIFT – is a cloud-based Kubernetes platform that helps developers build applications. It offers automated installation, upgrades, and life cycle management throughout the container stack — the operating system, Kubernetes and cluster services, and applications — on any cloud (liet. *yra debesų kompiuterijos pagrindu sukurta Kubernetes platforma, kuri padeda kūrėjams kurti programas. Ji siūlo automatizuotą diegimą, atnaujinimus ir gyvavimo ciklo valdymą visame konteinerių krūvoje – operacinėje sistemoje, Kubernetes ir klasterio paslaugomis bei programomis – bet kuriame debesų kompiuterijos centre..*).

POD – can be defined as a collection of containers and its storage inside a node of the OpenShift (Kubernetes) cluster (liet. *gali būti apibrėžtas kaip konteinerių rinkinys ir jo saugykla OpenShift (Kubernetes) klasterio mazge.*).

SOAP – messaging protocol specification for exchanging structured information in the implementation of web services in computer networks (liet. *pranešimų protokolo specifikacija, skirta keistis struktūrizuota informacija diegiant žiniatinklio paslaugas kompiuterių tinkluose.*).

SOLID – acronym that stands for five key design principles: single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle (liet. *akronimas, reiškiantis penkis pagrindinius projektavimo principus: vienos atsakomybės principas, atviro uždarymo principas, Liskovo pakeitimo principas, sąsajos atskyrimo principas ir priklausomybės inversijos principas.*).

Contents

Introduction

Problem Formulation

Microservice architecture is becoming the standard by default in most enterprises because many projects have been implemented using this architecture in the last few years, and the results have been very positive. Top companies, such as Amazon, eBay, Netflix, PayPal, Twitter, and others, successfully shifted from a monolithic to a microservice architecture.

Microservice architecture, as well as software development and IT operations (DevOps) practice, improve software development agility and flexibility. Enterprises can bring their digital products and services to a very competitive market faster. Microservice architecture is becoming a design standard for modern cloud-based software systems because it helps develop a cloud-native application. Using microservices and embracing cloud-native technologies is the way to reduce development time and increase deployment speed.

Migration from a monolithic architecture to a microservice architecture is a complex challenge that consists of issues such as microservice identification, code decomposition, communication establishment between microservices, data storage adaptation, independent deployment, etc. Extracting microservices from legacy monolithic software is an extremely complicated task. Each enterprise application is unique. It was programmed using different programming languages and techniques, and different databases and communication mechanisms were used; therefore, it creates different challenges. Different organisations use different migration patterns, techniques and methods because microservices are still a relatively new architectural approach that has no widely approved implementation methods.

Relevance of the Dissertation

According to the International Data Corporation, 89% of some 300 North American enterprise survey respondents already use microservices (Olofson et al., 2021; Anand, 2021). The International Data Corporation predicts that 90% of all new applications will be developed based on microservice architecture. To remain competitive, companies have started to modernise their legacy monolithic systems by decomposing them into microservices (Francesco et al., 2018; Knoche et al., 2018; Wang et al., 2020; Wolfart et al., 2021; Beni et al., 2019; Mohamed et al., 2021).

Although the topic of monolithic software migration into microservice architecture has already been explored by scientists and software engineers, it is a complex and relatively new challenge; therefore, little research exists on its many parts, such as database adaptation during the migration and communication establishment between microservices. The research primarily focuses on microservice identification within monolithic applications and source code decomposition into microservices. The author of this research proposes to bridge this gap using a migration approach that consists of three main parts: code decomposition methods, communication, and data management.

Research Object

The object of the present research is methods of migrating legacy monolithic applications to microservice architecture.

Aim of the Dissertation

This dissertation aims to improve migration from legacy monolithic applications to microservice architecture by proposing a novel migration approach that includes code base decomposition, communication establishment, and data management.

Tasks of the Dissertation

The following problems had to be solved to achieve the objective:

1. To review microservice architecture and existing techniques of legacy monolithic software migration into microservice architecture by conducting a literature review and identifying the most important aspects and existing gaps.
2. To investigate code decomposition methods of migration from legacy monolithic software into a microservice architecture.
3. To investigate communication technologies for microservices and determine particular cases for their use.
4. To propose and evaluate the approach of monolithic database migration into multi-model polyglot persistence based on microservice architecture.
5. Devise an approach grounded in meticulous analysis and experimental findings to effectively manage code decomposition, establish microservice communication, and handle databases during the transition from monolithic systems to microservice architecture.

Research Methodology

To investigate the *object*, the following *research methods* were chosen:

1. A systematic scientific literature review was conducted on existing techniques of legacy monolithic software migration into a microservice architecture. Strengths and weaknesses were summarised. Existing gaps in communication establishment and data management fields were identified.
2. The experimental research method was applied to investigate communication technologies for microservice architecture. The advantages and disadvantages of each technology were summarised, and particular cases of their use were determined. All microservices were written using the C# programming language. Latency tests were conducted using the BenchmarkDotNet library. Throughput tests were executed by using the NBomber library.
3. The constructive research method was employed to develop and validate the proposed approach for migrating a monolithic database into a multi-model polyglot persistence based on microservice architecture.

The ArangoDB database was used as the multi-model polyglot database engine. The microservice that exposes multi-model polyglot persistence was written using the C# programming language.

The Scientific Novelty of the Dissertation

The scientific novelty of this research is specified as follows:

1. The proposed migration approach from legacy monolithic software to a microservice architecture stands out in the realm of microservice migration by uniquely encompassing three essential components: code decomposition, communication establishment, and data management. This contrasts with conventional methods, which often provide more limited coverage by addressing only the code decomposition part.
2. The novel migration approach shifts monolith databases to a multi-model polyglot persistence within a microservices architecture. This transformation enhances consistency, understandability, availability, and portability while successfully preserving data quality across eleven of the ISO/IEC 25012:2008 standard attributes.
3. The proposed criteria offer a distinctive framework for selecting a code decomposition method from three available choices, each uniquely scrutinised across eight criteria, including microservice size and count. Additionally, the criteria provide an innovative basis for choosing among five communication technologies, evaluated and compared based on eight criteria, such as latency and throughput.

The Practical Value of the Research Findings

The proposed novel migration from legacy monolithic software to a microservice architecture approach allows for the execution of the migration based on three main aspects: code decomposition, communication establishment, and transformation of data management. By using the proposed migration approach, migration executors can choose one of three code decomposition methods and one of five communication technologies based on their needs. Research results showed that the proposed data management approach can be used to conduct data storage migration from a monolith to a microservice architecture and improve the quality of the consistency, understandability, availability, and portability attributes. Moreover, the author expects that his results could inspire researchers and practitioners towards further work aimed at improving and automating the proposed approach.

Defended Statements

The following statements based on the results of the present investigation may serve as the official hypotheses to be defended:

1. The proposed migration approach allows for an enhancement in areas of consistency, understandability, availability, and portability. The transition from a monolithic mainframe persistence model to a multi-model polyglot persistence model not only adeptly addresses these pivotal concerns but also excels in up-holding data quality, spanning eleven of the fifteen ISO/IEC 25012:2008 standard quality attributes.
2. RabbitMQ and gRPC are the most suitable technologies if latency and throughput are the main criteria for choosing a communication technology during the migration from a monolithic architecture to a microservice architecture. Binary serialisation used by gRPC outperforms RabbitMQ when communicating messages with higher complexity.
3. Code-based and storage-based methods allow for identifying technical functions and group code and storage components based on them, while business-domain-based methods allow the decommissioning of applications into microservices based on identified business domains. Microservices based on technical function provide higher granularity.

Approval of the Research Findings

The results of the dissertation were published in two scientific publications in reviewed scientific journals indexed in the *Clarivate Analytics Web of Science* database with Science Citation Index, and two were published in conference proceedings. The author gave four presentations at international scientific conferences:

– 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream) 1 April 2019, Vilnius, Lithuania.
– Baltic DB&IS 2020, 14th International Baltic Conference on Databases and Information Systems, 16–19 June 2020, Tallinn, Estonia.
– Data Analysis Methods for Software Systems (DAMSS), 2–4 December 2021, Druskininkai, Lithuania.
– 2023 Open Conference of Electrical, Electronic and Information Sciences (eStream) 27 April 2023, Vilnius, Lithuania.

The Structure of the Dissertation

The dissertation consists of an introduction, five main chapters, general conclusions, references, a list of publications by the author on the topic of the dissertation and a summary in Lithuanian. The total scope of the dissertation is 162 pages, one equation, 74 figures and 21 tables.

1

Analysis of Microservice Architecture and Methods of Migration from Legacy Monolithic Software into Microservice Architecture

This chapter reviews microservice architecture and its advantages and disadvantages over monolithic architecture. It starts by explaining the most important aspects of microservice architecture and the reasons why companies are aiming to migrate their legacy monolithic software to it. Next, the text provides an analysis of existing migration from legacy monolithic software to microservice architecture methods. It explains the difference between rebuilding and modernising. Different migration methods are analysed, and their advantages and disadvantages are provided. Different communication technologies, techniques and aspects are explained, and findings of the communication between microservices analysis are provided. Finally, a literature review is conducted of one of the key issues for microservice architecture: data storage adaptation to a microservice architecture.

Four publications were published on the topic of this chapter (Kazanavicius, Mazeika et al., 2019; Kazanavicius, Mazeika et al., 2020; Kazanavicius, Mazeika, Kalibatiene et al., 2022; Kazanavicius, Mazeika et al. 2023).

1.1. Microservice Architecture

Monolithic architecture is the traditional software development method when all functions are encapsulated into one single application. Monolithic software is designed to be self-contained. This type of architecture is tightly coupled, which means that if one of the components is not present, then it will not be executed or compiled. Monolith architecture has benefits and drawbacks. The following benefits of monolithic architecture can be mentioned: fewer cross-cutting concerns – it is simpler to hook up components to cross-cutting concerns when everything is running through the same application; less operational overhead – only one application needs to be set up, and less complex to deploy – only one application needs to be deployed. Drawbacks of monolithic architecture are as follows: coupled – it is especially difficult to make changes when monolith becomes highly complex; continuous deployment – the entire application should be deployed on each update; scalability – it is difficult to scale when different modules have conflicting resource requirements; and reliability – a bug in any component can potentially bring down the entire application (Dehghani et al., 2018; Fritzsch et al., 2018; Kalske et al., 2017).

Usually, legacy applications grow in size and complexity, leading to monstrous monolithic software after several years of development, and the disadvantages of monolithic architecture outweigh its advantages (Blanch et al., 2017). Fixing bugs and adding new features to such applications is a complex and time-consuming operation. Scalability is usually impossible or requires a lot of work. Under such circumstances, organisations start looking for a new architectural solution (Dehghani et al., 2018). Microservice architecture is becoming a standard by default in most enterprises because many projects have been implemented using this architecture in the last few years, and the results have been very positive. Top companies, such as Amazon, eBay, Netflix, PayPal, Twitter, and others, have successfully shifted to microservice architecture (Kwiecen, 2019).

A microservice architectural (Fig. 1) style is an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often HTTP resource API. These services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of centralised management of these services, which may be written in different programming languages and use different data storage technologies (Fowler et al., 2014).

The main three principles of microservice architecture are (Blinowski et al., 2022):

- *Microservice has a single responsibility:* similar to the single responsibility principle from SOLID principles, where every class should have only one responsibility. Multiple microservices should not share the same responsibility, and none of the single microservices should have more than one responsibility. Each microservice should deliver complete business capability as one unit. In other words, microservices should perform only one function.
- *Microservice is autonomous:* it is a self-contained and independently deployable service. Due to its autonomy, it must contain all dependencies, such as libraries and the execution environments – web servers, containers, virtual machines, etc.
- *Microservice is a polyglot:* it exposes its endpoints as APIs and abstracts all its implementation details, such as implementation logic, architecture, technologies, etc.

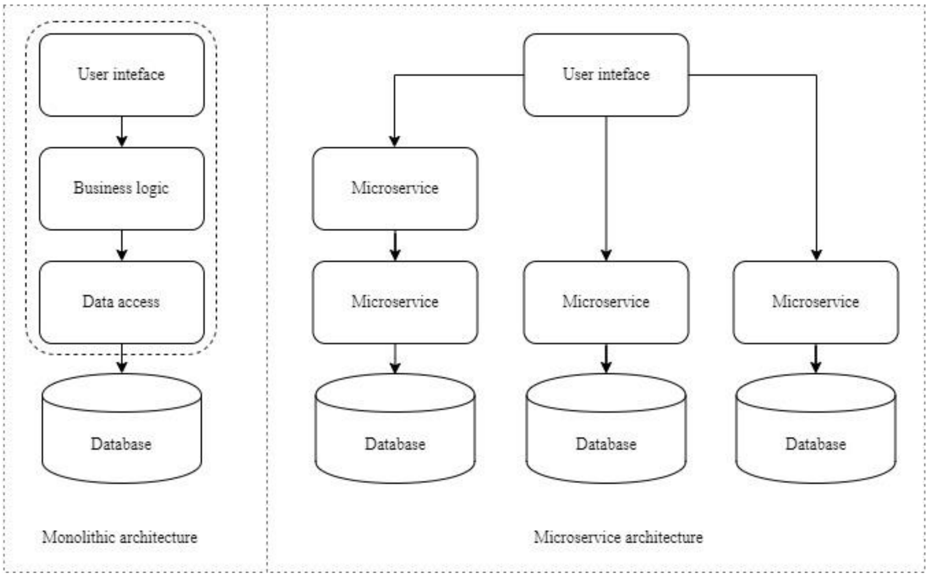


Fig. 1.1. Comparison of monolithic and microservice architectures

One of the main reasons why microservice architecture is considered a better option than monolithic architecture is the decomposition of complex applications into smaller components that are easier to develop, manage, and maintain than a single monolith application. Splitting applications into distinct, independent microservices allows individual teams to manage them within the software development organisation and work independently (Ghofrani et al., 2018). Because microservices are autonomous and communicate via open protocols, they can be developed independently with different technologies and programming languages (Al-Debagy et al., 2018; de Camargo et al., 2016; Lenarduzzi et al., 2018). Usually, teams developing microservices are organised around business rather than technical capabilities. Each new requirement should be addressed by only one microservice to retain independent development (Ghofrani et al., 2018; Atchison, 2018). Independence and autonomy allow microservices to be scaled horizontally, technically-wise, and within the organisation, as teams can be smaller and more agile. Consequently, microservice architecture improves technical aspects and increases business agility and the possibility of delivering new features faster (Blinowski et al., 2022; Lenarduzzi et al., 2020; Ramin et al., 2020).

Other worth mentioning benefits of microservice architecture compared to monolithic architecture (Pozdniakova et al., 2017; Chen et al., 2017; Blinowski et al., 2022):

- *Deployability*: microservices can be deployed independently, and there is no need to restart an entire application. The possibility of identifying critical business functionality allows the deployment of corresponding microservices in a more redundant environment.
- *Reliability*: a microservice's fault affects that microservice alone, not necessarily the entire application. Loosely coupled architecture makes microservices more fault-tolerant.
- *Cloudability*: the deployment characteristics make microservices a great match for the elasticity of the cloud. Microservices are cloud-native applications. Because microservices are independent processes, each could be deployed to a separate container or virtual machine in the cloud. Microservices could be updated and scaled separately. Scalability could be controlled by load requirements on demand. This approach enables more granular application elasticity. Solutions like Docker or Rocket containers, together with Docker Swarm, Mesos, or Kubernetes orchestration tools, enable microservice architecture to be used as architecture for cloud-ready applications.
- *Modifiability*: each microservice is encapsulated; therefore, it is more flexible to use new frameworks, libraries, data sources, and other resources. Management of the microservice-based application development is divided across smaller teams that work more independently. The microservice architecture allows for achieving better alignment of developers with business users since microservice architecture is organised

around business capabilities, and developers can easily understand user perspective and create microservices that are better aligned with the business needs.

The microservices architecture is not a panacea and has drawbacks (Chen et al., 2017; Blinowski et al., 2022). Complexity is its biggest drawback compared to monolithic architecture. Microservice architecture adds complexity to the project just by being a distributed system. Deployment, scaling, and monitoring are more complex tasks in microservice architecture than in monolithic architecture.

Both monolithic and microservice architectures have their advantages and disadvantages, and the choice between them depends on various factors such as the size of the project, the team's experience, the complexity of the system, the desired scalability, etc. It is necessary for developers and architects to closely evaluate whether the decomposition of an existing monolith is the right path and whether the microservices itself is the right destination (Dehghani et al., 2018).

A monolithic architecture better suited for a simple, lightweight application. The microservice architecture solution is the better choice for complex, evolving applications. Monolith applications should be modernised to a microservice architecture when:

- The monolithic application becomes too big and complex to maintain or extend. It becomes very expensive, both in terms of resources and time, to perform daily maintenance operations, add new functionality or fix existing issues.
- Modularity and decentralisation are important aspects. The microservice architecture allows working on each microservice separately. Challenges, such as scalability, can be applied only to a specific microservice instead of the entire application.
- Preference for gaining long-term benefits in comparison to those in the short term.

An environment that supports microservices fundamentally needs a set of baseline requirements to ensure some level of sanity. An organisation must be willing to bear the overhead of starting and supporting them. The overhead will not be insignificant. Well-performed microservices will take time and money. Each organisation must have an internal group responsible for infrastructure, which will be provided for development and operation teams to use microservices. This group must consist of the best organisation's developers or even external consultants. There is no one rule for setting up an infrastructure for microservices. Each area below describes functionality that should be implemented in infrastructure (Mayer et al., 2018).

Continuous Integration/Continuous Delivery: an organisation should decide how microservices should be built, tested and deployed. These operations should be automatic (Andrawos, 2018; Douglass, 2018). Nowadays, many build systems provide pipeline functionality (Balalaie et al., 2016; Levcovitz et al., 2016). The group responsible for microservice infrastructure should decide on a strategy for how to do it and choose tools for it:

- *Source control*: how the source code should be stored and maintained.
- *Build tool*: how the microservice should be built.
- *Tests tool*: how the tests should implemented.
- *Deploy tool*: how the microservice should be deployed.

Virtual Machines/Containers and Cloud: another important decision is what technology to use for the execution environment. Many enterprises with existing applications running on a stable virtual machine infrastructure are choosing to take a "toe in the water" approach. By deploying containers on virtual machines, they get the benefits of mature monitoring and isolation with more rapid DevOps processes. Compared to containers running on bare metal, they do give up some performance, scalability, and cost. But it is certainly a valid way to transition (Azarny, 2017). Microservice architecture is a natural fit for cloud-native applications. A cloud-native application is defined as an application built from the ground up for cloud computing architectures. This means that the application is cloud-native if it is designed as if it is expected to be deployed on a distributed and scalable infrastructure (Pozdniakova et al., 2017; Mulesoft, 2018).

Monitoring is a critical part of the infrastructure of microservices. Organisations should follow five principles to establish more effective monitoring, which are listed below. These principles will allow organisations to address both the technological changes associated with microservices and the organisational changes related to them (Rosendahl, 2016).

- Monitor containers and their content.
- Alert on service performance but not on container performance.
- Monitor services that are elastic and have multi-location.
- Monitor APIs.
- Map monitoring to organisational structure.

Logging plays a critical part in application maintenance. To do it efficiently for microservices, a logging service should be centralised and have a strong visualiser. Best practices for logging microservices are listed below (Dave et al., 2016; Soroko, 2017; Melendez et al., 2018).

- Correlate Requests with a Unique ID.
- Include a Unique ID in the Response.

- Structure Your Log Data.
- Add Context to Every Request.
- Write Logs to Local Storage.
- Log Useful and Meaningful Data to Avoid Regret.

1.2. Legacy Software Migration Methods

Migration to microservices from monolithic legacy software cannot be done fast. It is important to know that there is a high overall cost associated with decomposing an existing system into microservices, and it may take many iterations (Dehghani et al., 2018; Fowler et al., 2014). Because enterprise legacy application is a broad term, it is not possible to say that there is only one good way to migrate from legacy monolith to microservice architecture (Linthicum, 2018; Linthicum, 2017; Koltovich, 2017). Because microservices are a relatively new architectural style and no widely approved way of migrating exists, different organisations use different migration patterns and techniques (Furda et al., 2018; Mishra et al., 2018).

One of the key challenges in this context is the extraction of microservices from existing legacy monolithic code bases (Carrasco et al., 2018; Mazlami et al., 2017). This chapter reviews different techniques used to accomplish migration. In general, there are two strategies, e.g., rebuilding and modernisation (refactoring).

Not all monolithic applications can be easily migrated to microservice architecture. Sometimes, it is more economically beneficial to rebuild an application from scratch instead of refactoring it (Linthicum, 2018). The following type of legacy applications is not recommended for refactoring:

- Very old applications that are built using very old languages and databases that are not up to current standards.
- Applications that have a poor design.
- Applications that are tightly coupled to the database.

A different approach to legacy application modernisation is to refactor everything to split legacy apps into microservices and connect these microservices into one platform (Linthicum, 2018). Different ways to decompose legacy monolith applications into microservices are shown in Table 1.1. Each of them has benefits and drawbacks. Some of them are very general and could be used with any type of application, while others are more specific and will work only with some assumptions.

Table 1.1. Migration methods

Description of the migration method	
Genc Mazlami, Jurgen Cito and Philipp Leitner present a formal microservice extraction model to allow algorithmic recommendation of microservice candidates in a refactoring and migration scenario (Mazlami et al., 2017). The authors present a tool that supports structured service decomposition through graph cutting. The internal representation of the system to be decomposed is based on a catalogue of 16 different coupling criteria that were abstracted from literature and industry know-how. Software engineering artefacts and documents, such as domain models and use cases, act as input to generate the coupling values that build the graph representation. The evaluation was conducted on 21 open-source projects written in Java, Ruby, and Python programming languages.	
Benefits	Drawbacks
The performance evaluation shows that, for the most part, the proposed approach scales concern the size of the revision history (logical and contributor coupling). The quality evaluation shows that the proposed approach can reduce the microservice's team size to a quarter of the monolith's team size or even smaller.	One limitation is the fact that the extraction model is based on classes as the atomic unit of computation in the strategies and the graph. Using methods, procedures or functions as atomic units of extraction might potentially improve the granularity and precision of the code rearrangement and reorganisation.
Description of the migration method	
Rui Chen, Shanshan Li and Zheng Li proposed a top-down analysis approach and developed a dataflow-driven decomposition algorithm (Chen et al., 2017). The three-step process is defined below: • Engineers, together with users, conduct business requirement analysis and construct a purified while detailed dataflow diagram of the business logic. • The algorithm combines the same operations with the same type of output data into a virtual abstract dataflow. • The algorithm extracts individual modules of "operation and its output data" from the virtual abstract dataflow to represent the identified microservice candidates.	
Benefits	Drawbacks
A dataflow-driven mechanism guarantees the most fine-grained microservice candidates in terms of data operation within a business logic. Extraction can accept various text sources besides the webpage content, and it cares little about where its output data will go.	Identifying the same data operations requires expertise to some extent. Candidate microservices obtained from the suggested decomposition mechanism could still need expert judgment before being developed in practice. The proposed decomposition mechanism has not been widely applied to large-scale projects.

Continued Table 1.1

Description of the migration method	
Alessandra Levcovitz, Ricardo Terra and Marco Tulio Valente describe a technique to identify microservices on monolithic systems (Levcovitz et al., 2016). The evaluation was conducted on 750 KLOC programs written with C programming language and DBMS with 198 tables. The proposed technique consists of the following steps: • Database decomposition – map database tables into subsystems based on business functions. • Dependency Graph – create a dependency graph between facades, business functions and database tables. • Based on the dependency graph, identify pairs of facades and database tables. Map subsystems and identify pairs of facades and database tables. • Identify candidates to be transformed on microservices. For each distinct pair obtained in the prior step, inspect the code of the facade and business functions that are on the path from the façade to the database table in the dependency graph. • Create API gateways to make the migration to microservices transparent to clients. API gateway consists of an intermediate layer between client-side and server-side applications.	
Benefits	Drawbacks
The proposed technique was able to identify and classify all subsystems and create and analyse the dependency graph when evaluating and classifying only database tables into business subsystems, which demands access only to the source code and the database mode.	In some scenarios, an additional effort might be needed to migrate the subsystem to a set of microservices: subsystems that share the same database table. A microservice represents an operation that is always in the middle of another operation. Business operations that involve more than one business subsystem on a transaction scope.
Description of the migration method	
Zhamak Dehghani proposed a very formal migration process from monolith to microservice architecture (Dehghani, 2018). The suggested flow consists of these principles: • Minimise dependency back to the monolith. • Split sticky capabilities early. • Decouple vertically and release the data early. • Decouple what is important to the business and changes frequently. • Decouple capability and not code. • Migrate in atomic evolutionary steps.	
Benefits	Drawbacks
The migrating process with this approach can be divided into small steps. It is possible to safely stop and restore.	Very long migration process. It is a very formal way without any measurements.

End of Table 1.1

Description of the migration method	
Holger Knoche and Wilhelm Hasslbring proposed a migration process to decompose an application into microservices (Knoche et al., 2018). The evaluation was conducted on 100 KLOC programs written with Cobol programming language. The modernisation process consists of five steps: • Defining an External Service Facade – defining an external service facade that captures the functionality required by the client systems in the form of well-defined service operations. • Adapting the Service Facade – implement an external service façade functionality. • Migrating Clients to the Service Façade – start using newly created external service façade interfaces. • Establishing Internal Service Facades – restructure applications internally. • Replacing the Service Implementations with Microservices – monolithic application is replaced by microservices.	
Benefits	Drawbacks
Establishing well-defined, platform-independent interfaces based on the bounded contexts of the underlying domain. Reducing the number of entry points and preventing access to the internals, moving noncustomer functionality into separate components, and eliminating redundant and obsolete parts of the application.	Certain parts of the application cannot be modernised using the presented approach. In particular, some user interfaces, which are built on highly proprietary technologies, lack the necessary means for service abstraction.
Description of the migration method	
Chen-Yuan Fan and Shang-Pin proposed a migration process based on SDLC, including all of the methods and tools required during design, development, and implementation (Fan et al., 2017).	
Benefits	Drawbacks
Specialised and simple: microservices are designed to handle problems in a single domain. Fault Tolerance: one microservice's fault cannot break the entire application. Automated: automation tools used for building, deployment, and monitoring.	Complex environment settings: the configuration is not as simple as in a Monolithic architecture system, and many automation tools must be carefully set up to achieve the desired results. Using more resources: microservices use multiple tools to achieve architectural flexibility, such as Service Discovery and API Gateway.

1.3. Investigation of Methods of Migration from Legacy Monolithic Software into Microservice Architecture

Extracting microservices from legacy monolithic software is an extremely complicated task. Each enterprise application is unique. It was programmed using different programming languages and techniques, and different databases and communication mechanisms were used; therefore, it creates different challenges. During the literature review and analysis, three main directions on how decomposition from monoliths to microservices could be realised were identified: *Storage-based* – all code related to specific storage items like database or database table should be placed in one microservice. *Code-based* – application decomposition should be implemented based on code items like class. Application functions should be identified, and all code items should then be assigned to one of these functions. *Business-domain-based* – applications should be divided into microservices by business domains, for each business domain should be a separate microservice (Levcovitz et al., 2016; Mazlami et al., 2017; Fan et al., 2017; Chen et al., 2017; Knoche et al., 2018).

Three methods were chosen for the analysis because each is the best representation of a separate direction of how decomposition from monoliths to microservices could be implemented. Other methods found during the literature review and analysis use the same directions or combine them to achieve better results.

A comparison between selected methodologies was made by decomposing the same enterprise legacy monolith application into microservices three times, using all selected methodologies. The benefits and drawbacks of each methodology were analysed and compared.

1.3.1. Enterprise Monolithic Application Architecture

An enterprise legacy monolithic application named DataProvider was selected for this analysis because its functionality and architecture are very common in enterprise organisations, and its size allows it to conduct decomposition within a relatively short period (2–4 months). Although the system was not large and complex, it had a standard architecture and was composed of three components: API, database, and business logic. Due to its size and simplicity, it is perfectly suited to be a subject in comparison to selected methodologies. The dissertation's author posits that the acquired findings and deductions possess scientific merit, warranting application in the dissection of more extensive and intricate systems. The dissertation's author acknowledges that a nuanced decomposition tailored for larger and more complex systems would enhance the precision and depth of both results and conclusions.

The primary function of the DataProvider application is to provide important organisation's data from one place to other information systems within organization. An organisation stores different types of data, like accounts, books, customer data, etc., in different mainframe systems. The DataProvider application reduces complexity because fewer integrations are needed and increases performance because the mainframe is slower than the DataProvider.

The DataProvider application (Fig. 1.2) is written with Microsoft .NET framework, and C# programming language is used. It consists of three main components:

1. *Business logic* – collecting and caching data from old mainframe systems. Business logic writes collected data to the DataProvider local database.
2. *Database* – MS SQL database technology is used to store collected data from mainframe systems.
3. *Rest API* – HTML endpoint for other information systems to access important organisation's data in DataProvider. Swagger tools are used to provide Rest API functionality.

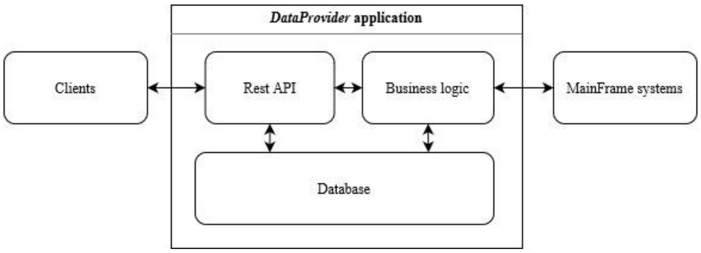


Fig. 1.2. DataProvider application architecture

The DataProvider application is a relatively small and simple typical enterprise application containing three main parts: UI, logic, and database. It has 350 classes, 5500 lines of code, 44 facades, and 15 database tables. More details about the code quality are presented in Table 1.2.

Table 1.2. DataProvider code quality

Parameter	Average	Max	Min
Maintainability Index	86.2	100	40
Cyclomatic Complexity	8.6	116	0
Depth of Inheritance	1.5	5	0
Class Coupling	12.2	96	0

86.2 maintainability index and 8.6 cyclomatic complexity values indicate high code quality regarding maintainability. 1.5 depth of inheritance value shows that inheritance is widely used in the application. 12.2 class coupling value is high, which means that classes are coupled. High coupling is difficult to maintain and reuse. Code metrics values were obtained by using Visual Studio IDE (Microsoft, 2022).

1.3.2. Investigation Criteria of Methods of Migration from Legacy Monolithic Software into Microservice Architecture

This chapter provides information about criteria that were considered while investigating different methods of migration from legacy monolithic software into microservice architecture. As each legacy monolithic application could be different in many aspects, the list of criteria was introduced to compare migration methods from different angles:

- *Microservice candidate count*: to evaluate the potential number of microservices identified during the migration to microservice architecture within legacy monolithic applications.
- *Size of microservice*: to evaluate the potential size of extracted microservice from a legacy monolithic application.
- *The database*: to evaluate if the migration method supports monolithic database adaptation to microservice architecture.
- *Connecting microservices*: to evaluate if the migration method supports communication establishment between decomposed microservices.
- *The automation*: to evaluate the migration method’s possibility of being fully automated.
- *The technological stack*: to evaluate the technological stack used during the migration from monolithic architecture to microservice architecture.
- *Implementation and tools*: to evaluate implementation details and tools used during the migration from monolithic architecture to microservice architecture.
- *Code quality*: to evaluate the impact of code quality on the migration from monolithic architecture to microservice architecture.

1.3.3. Storage-Based Extraction Evaluation

Alessandra Levcovitz, Ricardo Terra and Marco Tulio Valente describe a technique to identify microservices on monolithic systems. The main idea of the technique is that decomposition should be done based on unique database tables and facade pairs. Each unique pair could be considered a microservice candidate. All business functions, which are used by a facade and database table pair, should be included in a microservice. During the decomposition process, facades, business functions, and database tables need to be identified, and unique pairs must be found. The proposed technique consists of the following four steps.

1.3.3.1. Database Decomposition

The first step is mapping database tables into subsystems. Each subsystem represents an organisation's business area. Tables unrelated to a business process called the control subsystem. Fig. 1.3 presents part of the database decomposition done in the DataProvider application.

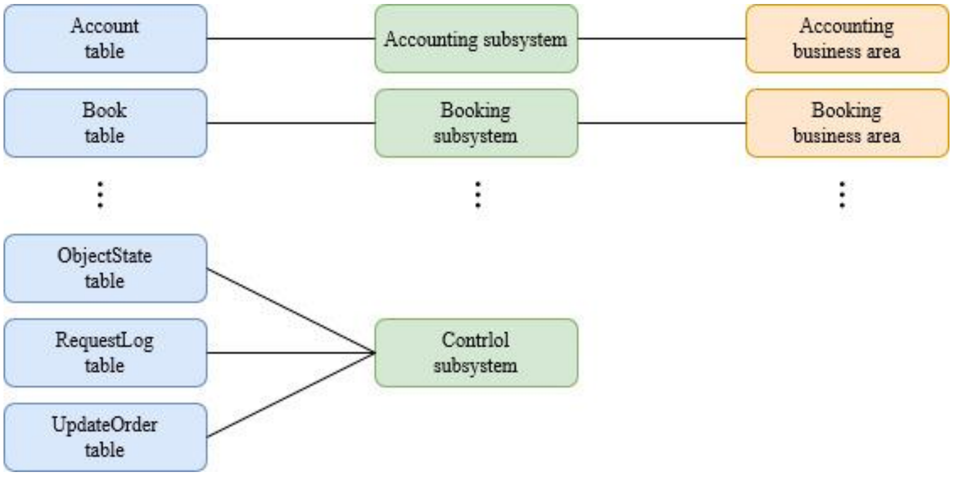


Fig. 1.3. Database decomposition

The DataProvider application has 15 database tables, nine subsystems, and eight different business areas. This step of the methodology allows for the identification of a number of tables and business areas. Identifying database tables is a task that requires only technical skills. On the other hand, identifying business subsystems and assigning a table to them require additional effort to understand the business process.

1.3.3.2. Dependency Graph

In the second step, a dependency graph between facades, business functions, and database tables was created. It shows business functionality and database dependencies. Fig. 1.4 illustrates some graphs of the DataProvider application.

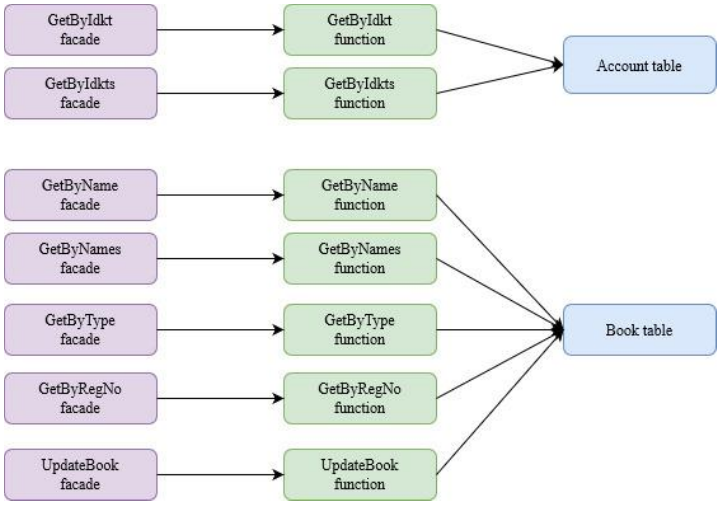


Fig. 1.4. Dependency graph

Five graphs were pretty straightforward: containing only one database table, one business functionality layer, and 0 dependencies from other database tables and business functionality subsystems. The other 12 database tables were joined into one more complex and complicated dependency graph. Some business functionality contains up to four dependencies from other database tables. Mostly, four business functionality layers were identified for full operation from the facade to the database table.

1.3.3.3. Database Tables and Facades Pairs

Based on the dependency graph, unique pairs of facades and database tables were identified and mapped with business subsystem functions. Fig. 1.5 presents two unique pairs in the DataProvider application.

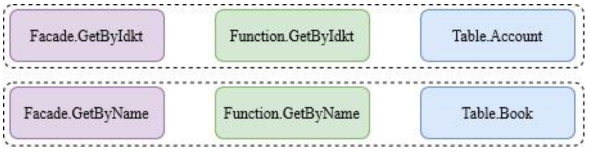


Fig. 1.5. Tables and facades pairs

The DataProvider application has 68 unique pairs of database tables and facades. Fifteen facades were in pairs with only one database table. Some of the facades were in pairs with different database tables up to eight times. More unique pairs with the same facade exist in a more complicated dependency graph.

1.3.3.4. Microservice Candidates

In the last step, candidates to be transformed into microservices were identified. For each distinct pair obtained in the prior step, inspection focused on the code of the facade and business functions that are on the path from the facade to the database table in the dependency graph. Fig. 1.6 illustrates two candidates to be transformed into microservices of the DataProvider application.

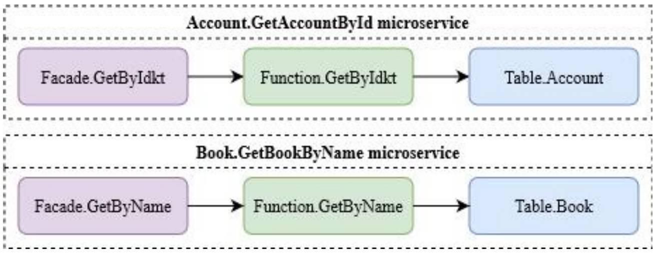


Fig. 1.6. Microservices candidates

The decomposition using a storage-based method resulted in 37 microservices candidates found in the DataProvider application. Detailed evaluation results are presented in Table 1.3.

Table 1.3. Storage-based extraction evaluation results

Subsystems	Tables	Functions	Microservice candidates
Accounting	1	2	2
Booking	1	5	5
Departments	1	3	3
Customers	5	9	15
Ratings	1	4	4
Users	3	3	3
Country definitions	1	2	2
Currency definitions	1	3	3

More functions and table subsystems had more microservice candidates identified. It is possible that the microservice candidate size could be very small if it contains only one business function. The method requires identifying business subsystems. To do so, business knowledge is needed, which is why the method's implementation could not be completely automated.

1.3.4. Code-Based Extraction Evaluation

Genc Mazlami, Jurgen Cito and Philipp Leitner created a model for microservice extraction from monolithic systems. The extraction model is based on code classes and their relationships. The application could be represented as a graph of its code classes. Decomposition is done by splitting a graph into microservice candidates. Which classes should belong to the microservice candidate could be determined by relationship weight. A higher weight value indicates stronger coupling. Microservices extraction from the monolithic systems model comprises three extraction stages: monolith, graph, and microservices. Two transformations take place between the stages.

1.3.4.1. Construction Step

The first step is the monolith transformation into the graph representation. In the graph, each vertex represents a class from the monolith and undirected edges represent its coupling with other classes in the monolith. Fig. 1.7 illustrates the construction step.

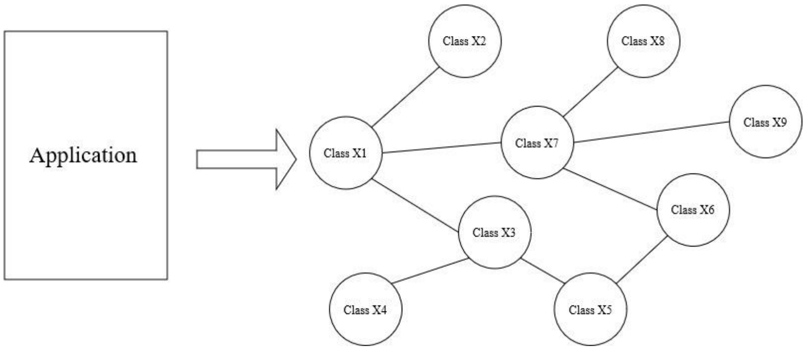


Fig. 1.7. Construction step

The DataProvider application has 273 classes. One class has the biggest number of dependencies, which is 96, and 17 classes have 0 dependencies and are not part of a graph. The average class coupling is ~10. Unit, integration, and manual test classes were excluded from the graph.

A better-quality code has fewer coupled classes, so a lower number of edges in the graph indicates a higher quality of the code. It is not clear how to treat class inheritance from the article; in this evaluation, a decision was made not to treat class inheritance as a dependency. Visual studio provides tools for extracting the information on code metrics automatically. It saves a lot of time in the construction step.

1.3.4.2. Clustering Step

The second and final transformation is to cut the graph into components that represent recommended microservice candidates. For this, the authors proposed three different strategies: logical coupling, semantic coupling, and contributor coupling. During this comparison, semantic coupling was chosen for evaluation.

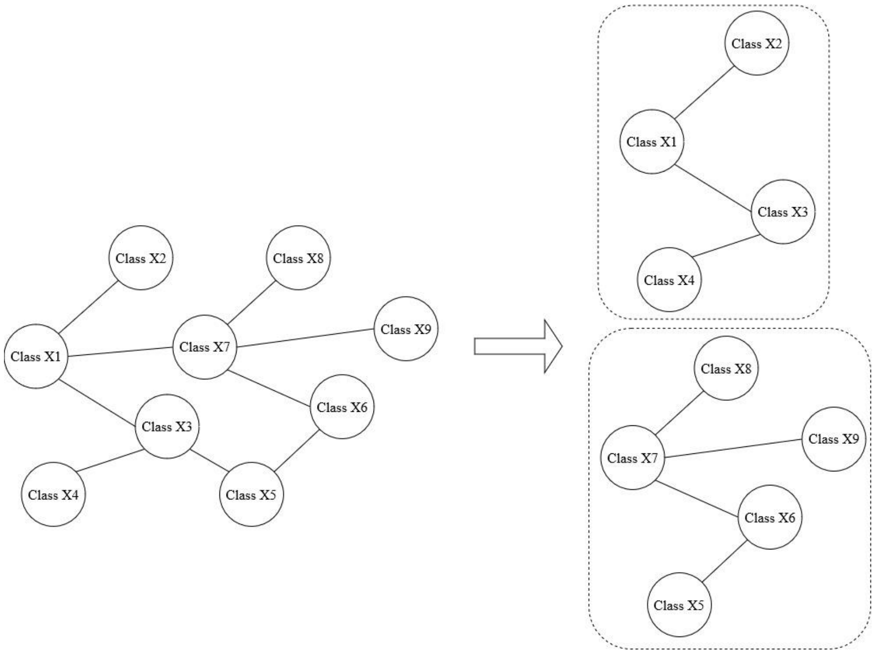


Fig. 1.8. Clustering step

The main idea of semantic coupling is that each microservice should correspond to one single defined bounded context from the problem domain. The strategy couples together classes that contain the code about the same “things”, e.g., domain model entities. Fig. 1.8 illustrates the microservice candidates’ extraction from the graph.

Eight microservices candidates were found in the DataProvider application. In total, 180 classes were identified for a specific business domain by class name. It was not possible to identify the business domain by class name for 93 classes.

About 33% of classes need an additional effort to be reviewed and assigned manually to the specific business domain or refactored and split into more classes. Detailed results of evaluations are presented in Table 1.4.

Table 1.4. Code-based extraction evaluation results

Business domain	Classes	Microservice candidates
Accounting	13	1
Booking	13	1
Departments	14	1
Customers	63	1
Ratings	13	1
Users	38	1
Country definitions	13	1
Currency definitions	13	1

Code quality plays a vital role in how easily a microservice candidate can be identified in the graph. If the code is written following clean code standards, the class should only have one responsibility, few dependencies, and a meaningful name. Automation can be accurate in extraction only if the monolith code is high quality. If a class has a lot of dependencies, no meaningful name or too many responsibilities, it is not clear to which microservice candidate it belongs. In this case, the additional effort is needed to refactor the class.

The code-based method is very formal and requires additional tools to be implemented properly. These tools are not available; only an algorithm and a mathematical model are provided, so organisations should implement them themselves.

1.3.5. Business-Domain-Based Extraction Evaluation

Chen-Yuan Fan and Shang-Pin proposed a migration process based on SDLC, including all of the methods and tools required during design, development, and implementation. The main criteria for a microservice candidate is the business domain; each separate business should have separate microservices. The proposed method suggests how specific business domain codes and database tables could be extracted. Two analysis methods are used in the migration of a legacy monolithic architecture into a microservice architecture.

1.3.5.1. Domain-Driven Design Analysis

In the first step, Domain-Driven Design (DDD) was used to find microservice candidates in the original system. The bounded context analysis results are a key tool for identifying microservice candidates in applications. The DDD was used to identify specific domains in the solution and identify domain modules in each domain. DDD approach analysis allows for the extraction of low-coupling microservices.

Eight different specific business domains were identified in DataProvider during DDD analysis: Accounting, Booking, Departments, Customers, Ratings, Users, Country definitions, and Currency definitions.

This step does not require any technical skills, only business process knowledge. It is a possibility that different people could identify different business domains per application. The business process tends to change in enterprise organisations so it’s possible that the business domain could change during migration from monolith to microservices.

1.3.5.2. Database Analysis

The second step involves the analysis of the database structure. It is common practice for each microservice to use a discrete database. This allows for avoiding high coupling between services. Foreign keys could be used as an indication of the microservice candidate.

The database schema of the DataProvider application could be divided into eight business domains identified in the DDD analysis stage. The customer business domain contains the biggest number of tables, e.g., five, and six business domains contain only one table.

1.3.5.3. New Architecture

The Domain-Driven Design and database analysis resulted in eight microservice candidates being found in the DataProvider application. One additional microservice should be created. Fig. 1.9 illustrates the new microservices architecture.

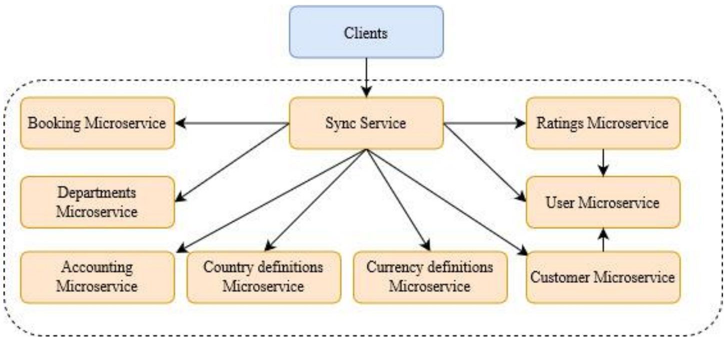


Fig. 1.9. Microservices extracted from DataProvider

To connect all microservices into one solution, a new microservice was introduced. Sync Service provides data synchronisation and an interface for front-end systems. Detailed results of evaluations are presented in Table 1.5.

Table 1.5. Business-domain-based extraction evaluation results

Business domain	Tables	Microservice candidates
Accounting	1	1
Booking	1	1
Departments	1	1
Customers	5	1
Ratings	1	1
Users	3	1
Country definitions	1	1
Currency definitions	1	1

The most important things for a successful migration from monolith to microservices using the business-domain-based method are strong business knowledge, business process stability in the organisation, and high-quality database schema.

1.3.6. Comparison of Migration Methods

This chapter compares different aspects of the evaluation results of extraction methods. *Microservice candidate count* and *Size of microservice* chapter overview how big and how many microservice candidates a method can extract. *The databases* chapter evaluates if methods can decompose databases within the monolith decomposition process. *Connecting microservices* analyses how microservices should work as one solution after the decomposition process. *The automation* chapter evaluates each method's possibility to be fully automated. *The technological stack* and *Implementation and tools* chapters provide more detail about how methods could be implemented and what technologies and tools could be used in the implementation. The last chapter, *Code quality*, evaluates the impact of the code quality in the decomposition process.

1.3.6.1. Microservice Candidates Count

The storage-based extraction method found most microservice candidates in the DataProvider application. The storage-based method found 37 candidates, the code-based method found eight candidates, and the business-domain-based method also found eight candidates.

In the storage-based method, microservice is extracted as a concrete function in the application while in the business-domain-based method, microservice represents a specific business domain. The storage-based method will always provide more microservices than the business-domain-based method because the business domain always has at least one function.

The code-based method is more flexible than other compared methods. It provides the optionality to choose a strategy for how microservice should be extracted. A semantic coupling strategy was chosen during this comparison. Its key idea, in general, is very similar to Domain-Driven Design, which explains why it found the same number of microservice candidates as the business-domain-based method. Another extraction strategy is logical coupling, which focuses on concrete functions. It could be predicted that microservice candidates were similar to Method I. The last strategy is contributor coupling, the main idea of which is that microservice should belong to one team. In this case, the number of microservice candidates directly depends on the number of teams working with an application.

1.3.6.2. Size of Microservice

The main idea of a microservice is that it should have only one responsibility. The technical community interprets it differently. What kind of responsibility? Is it a Business or functional type? Business responsibility is bigger than a function because it contains at least one function and usually much more than one. Split by functions, microservices are much smaller and have been named serverless.

Suppose organisations decide to split their monolith application into microservices by business domains. Then, they should choose the business-domain-based method or the code-based method with a semantic coupling strategy. If the decision is to split into microservices by functions, the storage-based method or code-based method with a logical coupling strategy could be used. The real difference in a microservice’s size depends on how much the business domain contains functions. The more functions the business domain will have, the bigger the microservices candidate will be extracted using the business-domain-based method or the code-based method with a semantic coupling strategy.

Given the absence of a universally agreed criterion for the optimal scale of a microservice, assessing the efficacy of a method remains inconclusive based solely on the number of microservices derived. The singular widely embraced guideline dictates adherence to the single responsibility principle. An organisation should delineate the designated scope of responsibility within a microservice and subsequently select the most fitting decomposition method to attain the desired outcome. The author proposes considering the quality attributes outlined in the ISO/IEC 25012:2008 standard as a viable approach for determining optimal outcomes.

1.3.6.3. Databases

The most common and popular practice is that each microservice should use its private database. The business-domain-based method fits this approach perfectly. After the Domain-Driven Design analysis, tables from the monolith database should be grouped and split into separate databases.

The storage-based method splits the monolith into microservices by functions, and some functions will most likely use the same table. If the decision was made to use this method, the database will probably be shared. Method authors do not provide any recommendations on how to deal with this challenge.

Code-based method authors assume that monolith applications use a repository pattern, and each table is represented as a repository class in the solution. Methods do not contain any recommendations on how databases should be adapted to the microservice architecture. A semantic coupling strategy approach used in the business-domain-based method could be used to split the database into separate databases for each microservice.

1.3.6.4. Connecting Microservices

To provide the same business value for users as the monolith application, microservices should be connected into one solution via lightweight mechanisms, often an HTTP resource API.

The storage-based method and the business-domain-based method suggest creating API gateways to make the migration to microservices transparent to clients. API gateway should be an intermediate layer between client-side and server-side applications. It handles requests from the client side using the same technology as it did before migration.

The code-based method does not provide any recommendations on how microservices should be connected after migration from the monolith architecture.

1.3.6.5. Automation

The code-based method with a contributor coupling strategy could be implemented fully automatically. The monolith must be implemented with object-oriented programming language because the extraction model is based on classes such as the atomic unit of computation and the graph.

The code-based method with a semantic coupling strategy could be implemented semi-automatically. In this case, business domains should be identified manually. How accurately the method will be able to identify the class relation to the business domain depends on the naming convention in the code.

The storage-based method and the business-domain-based method cannot be implemented automatically. The storage-based method requires manually identifying business subsystems and assigning database tables to one of the subsystems. The business-domain-based method requires two manual analyses.

1.3.6.6. Technological Stack

The storage-based method is designed to work with backend-type applications. It is programming language agnostic. Database storing data in tables must be part of the application because extraction uses tables to generate graphs.

The code-based method is suitable for backend-type applications written with object-oriented programming language. The extraction model is based on classes as the atomic unit. If the application is written in another type of language or several different languages, it is not possible to use a Code-based method for microservices extraction. Only one requirement exists for databases: repository pattern should be used in the code to describe database data models. SQL and NoSQL databases could be used.

The business-domain-based method is technologically agnostic and could be used with any kind of programming language and databases.

1.3.6.7. Implementation and Tools

The business-domain-based method is the least formal and most universal. On the other hand, it is most uncertain and requires the implementer to have a strong knowledge of the application business domain and implementation technical details.

The code-based method is the most formal and requires an additional tool to generate a graph representing the dependencies of classes. It is not clear what would be cheaper in terms of time and resources: implement a tool and use it or use other methods to do a microservice extraction from the monolith.

The storage-based method does not require any additional tools to implement, but it requires some knowledge of business domains to identify business subsystems. The storage-based method is less formal and more universal than the code-based method; on the other hand, the storage-based method is more formal and less universal than the business-domain-based method.

1.3.6.8. Code Quality

Code quality has the most impact on the code-based method because it creates a graph of the dependency classes. Clean and solid code generates more accurate graphs. A more accurate graph allows for the extraction of more accurate micro-services. Also, higher quality code is more readable, reusable, and transferable quickly.

The code quality also impacts the storage-based method and the business-domain-based method. The better the code quality, the easier it is to extract functions from it.

1.4. Communication

One of the biggest challenges while migrating from a monolith architecture to a microservice architecture is to define a proper communication technology. In monolith applications, communication between components is performed using process methods or function calls, while different communication methods have to be established to achieve the same functionality in a microservice architecture. A microservice-based application is a distributed system running on multiple processes or services. Therefore, microservices must interact using inter-process communication technologies. The design of communication between micro-services is one of the most significant challenges while migrating from monolithic software to microservices architecture (Microsoft, 2020).

1.4.1. Communication Technologies

Microservices can communicate in different ways, but all of them can be classified into two groups – synchronous and asynchronous. The client sends a request and waits for a response from the service in a synchronous communication style. It results in tight runtime coupling because both the client and service must be available for the duration of the request. Usually, HTTP/HTTPS protocols are used for synchronous communication. The main advantage of this communication is that the system is simple and easily implemented. Also, there is no intermediate component, such as a message broker. In asynchronous communication, micro-services communicate by exchanging messages over messaging channels based on advanced message queuing protocol (AMQP). All counterparts can send messages, and senders do not wait for the response message. There are several different asynchronous communication patterns, such as request–response, publish–subscribe, and notification. Loose runtime coupling and improved availability are benefits of asynchronous communication. However, its implementation is more complex. Message-based technologies, such as RabbitMQ, Apache Kafka, etc., use asynchronous communication between microservices. The most popular communications technologies used for microservices are based on HTTP protocol and asynchronous message patterns (Fowler et al., 2014; Microsoft, 2020; Bandhamneni, 2018; Galbraith, 2019).

The gRPC is an open-source Remote Procedure Call (RPC) framework developed by Google. It enables the establishment of transparent communication between server and client applications in any environment. Before gRPC became open source in March 2015, it had been used as a single general-purpose RPC infrastructure to connect a large number of microservices running within and across Google data centres for over a decade (Biswas et al., 2018; gRPC, 2022).

GraphQL is a query language for APIs and a runtime for filling those queries with existing data. GraphQL was developed internally by Facebook in 2012 and was published to the community in 2015. The key functionality of the GraphQL framework is a query language that allows clients to define the structure of the data required, and the same structure of the data is returned from the server (Hartig et al., 2017; Brito et al., 2020; Bandhamneni, 2018; GraphQL, 2022).

It must be noted that it is a common practice to use several communication technologies to develop microservice-based applications.

1.4.2. Architecture Patterns

Taibi et al. (2020) conducted a systematic literature review and identified three microservice orchestration architecture patterns that also include communication and coordination of the microservices. Patterns were classified as API Gateway, service discovery, and hybrid. A summary of the advantages and disadvantages of each architectural pattern was presented in the paper as well.

The API Gateway operates as the entry point of the system that routes the requests to the appropriate microservices. This pattern is technology agnostic but is usually implemented using the HTTP protocol. Ease of extension, market-centric architecture, and backward compatibility are the advantages of the API Gateway. The high complexity of implementation, low reusability, and low scalability can be mentioned as disadvantages of the pattern (Taibi et al., 2020; Montesi et al., 2016).

The service discovery pattern uses a different approach, e.g., the client can communicate with each service directly without an intermediate layer. The domain name system (DNS) address resolution into internet protocol (IP) address must be supported to achieve end-to-end communication between services. The pattern relies on the service-register service that performs similarly to DNS. The advantages of service discovery patterns are ease of development, maintainability, migration, communication, and health management. Disadvantages of the pattern are high coupling between the client and the service registry, high complexity of the service registry, and high complexity of the distributed system (Taibi et al., 2020; Montesi et al., 2016).

The hybrid pattern combines the service registry and the API gateway and replaces the API gateway with the message bus. Clients communicate only with the message bus that operates as a registry and gateway. Services communicate with each other via message bus, and direct communication between microservices is not used. The advantages of the pattern are ease of migration, while the disadvantages are high coupling between services and message buses (Taibi et al., 2020).

1.4.3. Streaming and Distributed Cache

Smid et al. (2019) discussed the balance between performance and coupling and pointed out situations where suggested architectures were appropriate. The authors introduced a streaming platform based on the message bus (Kafka) and data change capture platform (Debezium) to synchronise data between different databases effectively. Streaming is a different approach to orchestration and communication patterns mentioned in the previous chapter. The service-generating event notifies other services by using streaming events on the message bus. Therefore, almost all communication is performed by consuming events from the message bus or database. The proposed solution has a limitation: the overhead for deployment and maintenance for applying the streaming platform. The microservices need to be synchronised under a data model similar to the master system, and additional source code must be introduced. A distributed cache was introduced to improve communication performance. The advantages of using a distributed cache are performance, scalability, and ease of migration, while high complexity is a disadvantage. Communication performance decreases significantly when data changes frequently. The authors concluded that the message broker is an efficient way of communication between microservices, and the publish/subscribe model is very flexible and provides a faster mechanism than HTTP request with the benefit of persistent messages (Smid et al., 2019; Montesi et al., 2016).

1.4.4. Microservices and Service-Oriented Architecture Communication

Cerny et al. (2018) performed a detailed research analysing differences between microservice architecture and SOA. Microservices provide decomposition, preferring smart services while considering simple routing mechanisms without the global governance notable in SOA. This leads to higher service autonomy and decoupling since services do not need to make agreements on the global level. In general, there are two well-defined approaches used to coordinate services, e.g., using a central orchestrator or a decentralised distributed way. The centrally orchestrated approach is the typical SOA pattern, while the distributed approach is dominant for microservice-based applications. These approaches are named orchestration and choreography, respectively. Service orchestration works as a centralised business process, coordinating activities over different services and combining the outcomes. The choreography works without a centralised element. The control logic is described by message exchanges and rules of interactions as well as agreements among interacting services (Cerny et al., 2018; Smid et al., 2019).

1.4.5. Communication Security

Yarygina et al. (2018) analysed security challenges in a microservice architecture. Potential threats in microservice communication were identified, such as attacks on the network stack and protocols and attacks against protocols specific to the service integration style (SOAP, REST Web Services). Security threat mitigation techniques were proposed. The authors highlighted the leading microservice security industry practices, such as Mutual Authentication of Services using Mutual Transport Layer Security and Principal Propagation via Security Tokens. The authors proposed a method that combined both techniques and presented proof-of-concept evaluation results. Walsh et al. (2017) introduced new comprehensive, automated, and fine-grained mutual authentication mechanisms. To ensure a secure connection between microservices, the authors suggested using a combination of authentication and attestation. The proposed attestation mechanisms were built on top of standard transport layer security channels and certificates.

1.4.6. Communication Performance

Hong et al. (2018) provided a detailed research on the performance evaluation of RESTful API and RabbitMQ for Microservice Web Applications. Experimental results showed that when a large number of users sent requests to the web application in parallel, RabbitMQ, as the message-oriented middleware, provided more stable results compared to the RESTful API. On the other hand, the RESTful API has shown better request–response performance results.

Fernandes et al. (2013) performed a comparison research between a RESTful Web service and the AMQP protocol for exchanging messages between clients and servers. The final results showed that for applications that exchange a large amount of data, the best approach is to use the RabbitMQ server and the back-end service to consume the messages, process them, and send them to the database. As a result, fewer messages per second were sent, the time for exchange increased, and even more resources were used evaluating RESTful Web service.

It can be summarised that different factors like request load, IT environment, and network technologies determine communication performance between microservices. It cannot be unambiguously defined which communication technology is faster. It depends on the specific application. Asynchronous communication is a more robust and stable communication mechanism than HTTP (Rest) and enforces microservice autonomy. Detailed analysis and in-depth evaluation of chosen communication technologies are provided in the fourth chapter.

1.5. Data Management

Migration from a monolithic architecture to a microservice architecture is a complex challenge that consists of issues such as microservice identification, code decomposition, combination 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 a microservice architecture, each microservice works independently and has its private data storage, e.g., data storage is decentralised. 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.

Although the topic of monolithic software migration into microservice architecture has already been explored by scientists and software engineers, there is little research on database adaptation during the migration from a monolith to a microservice architecture. Despite this, it is recognised that data management is a major challenge in microservices (Laigner et al., 2021; Azevedo et al., 2019; Richter et al., 2017; Francesco et al., 2017; Knoche et al., 2019; Luz et al., 2018; Soldani et al., 2018). The primary focus of most of the research is microservice identification within monolith applications and source code decomposition into microservices. All of the existing migration methods provide very little to no recommendations on how to adopt data storage to a microservice architecture during the migration from a monolith to a microservice architecture. To the best of the authors' knowledge, besides Levcovitz et al. (2016), 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.

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

1.5.1. Structured Query Language versus Non-Structured Query Language

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. Inspired by SQL limitations, NoSQL has emerged as a solution to fill these gaps (Brewer, 2000; Khine et al., 2019).

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 (Khine et al., 2019).

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 (Meier et al., 2019). While SQL primarily focuses on consistency, NoSQL is giving up either consistency or availability and embracing partition tolerance (Brewer, 2000). There is no perfect database that could solve all the problems and fit 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, and 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 are 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 that is uniquely identified by an indexed key. Indexing and querying based on values are not supported, so in cases where querying is needed, it must be implemented on the client’s 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. 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 representing the relationships between them. The graph datasets are stored efficiently to provide effective operations for querying and analysis. 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 different types of data (Meier et al., 2019; Shah et al., 2016; Richter et al., 2017; Davoudian et al., 2016; Krishnan et al., 2002; Luz et al., 2018; Sharma et al., 2012; Nayak et al., 2013).

According to Nayak et al. (2013), NoSQL has several advantages: it provides a wide range of data models to choose from, is easily scalable, no database administrators are needed, it can handle hardware failures, it is faster and more flexible, and evolves at a very high pace. The disadvantages of NoSQL are its immaturity, inexistence of a standard query language, incompliance of some NoSQL databases with ACID, inexistence of a standard interface, and difficult maintenance.

1.5.2. Polyglot Persistence

The general polyglot persistence conception was evaluated from the point of view of 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, and 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 (Zdepski et al., 2018; Serra, 2015).

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 must 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 (Brewer, 2018; Khine et al., 2019; Meier et al., 2019; Shah et al., 2016; Zdepski et al., 2018; Zdepski et al., 2018; Wiese et al., 2015).

In 2012, Fowler predicted that polyglot persistence would occur over the enterprise as different applications use different data storage technologies. It would also occur within a single application as different parts of an application’s data store have different access characteristics. A hypothetical example of polyglot persistence is shown in Fig. 1.10. In the provided example, different types of databases are used to store different types of data to fulfil the concept of choosing the right tool for the given task (Fowler et al., 2012).

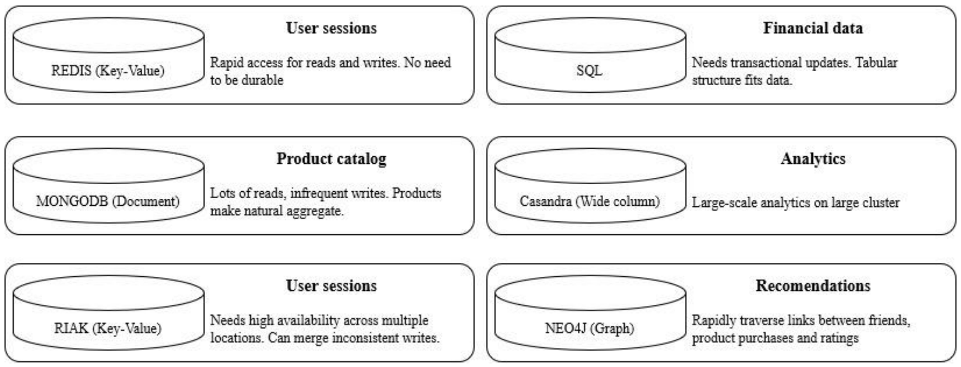

User sessions
REDIS (Key-Value)
Rapid access for reads and writes. No need to be durable
Financial data
SQL
Needs transactional updates. Tabular structure fits data.
Product catalog
MONGODB (Document)
Lots of reads, infrequent writes. Products make natural aggregate.
Analytics
Casandra (Wide column)
Large-scale analytics on large cluster
User sessions
RIAK (Key-Value)
Needs high availability across multiple locations. Can merge inconsistent writes.
Recomendations
NEO4J (Graph)
Rapidly traverse links between friends, product purchases and ratings

Fig. 1.10. Hypothetical example of polyglot persistence (Fowler, 2012)

On the other hand, polyglot persistence is a complex solution and creates many new challenges. A decision is required on which technology to use rather than just storing everything in one database. The immaturity of NoSQL tools is another issue. The consistency problem in an organisation raises the question of how to ensure data sync between different parts of the organisation.

Wiese (2015) categorised polyglot database architectures into three types: polyglot persistence, lambda architecture, and multi-model databases. Wiese (2015) recommends using polyglot persistence only if several diverse data models must be supported; otherwise, there is a risk of 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 accessing applications (Wiese et al., 2015).

Zdepski et al. (2018) proposed a modelling 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. 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 realises the translation of the conceptual model to the internal model of a database management system. The physical design implements all the peculiarities of each database software.

According to Shah et al. (2016), a crucial part of the efficiency of a polyglot system is the selection of a database engine (Shah et al., 2016). 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), (2) key–value type (Redis), and (3) a relational database (SQLite).

Trivedi et al. (2020). proposed the design of a polyglot persistence system for an e-commerce application based on the intelligent data mapper. A crucial part of the proposed design is the selection of databases: different databases are optimal for handling different types of data. Mapping of data from these dissimilar databases is only possible if the compatibility criteria are met. The proposed design consists of three types of databases: (1) document type (MongoDB) to store tore product details, customer details and other document-type data, (2) key–value type (Redis) to store data, such as product search counter, which requires constant update or modification, (3) relational database (SQLite) to store aggregate queries, such as payment details. The proposed polyglot persistence system was compared with a system where data was stored only in MongoDB and a system where data was stored only in SQLite. The most optimum results were obtained from the polyglot system.

An evaluation of the NoSQL multi-model data stores in polyglot persistence applications were conducted by Oliveira et al. 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.

1.5.3. 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 centralised management of services that may be written in different programming languages and use diverse data storage technologies (Newman, 2019).

In the book Building Microservices Applications on Microsoft Azure, Chawla et al. (2019) discuss the various critical factors of designing a database for microservice architecture-based applications. 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 the 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 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 Chawla et al. (2019), 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. To ensure that changes are efficiently transferred across the microservices, the authors suggest using two approaches: (1) a two-phase commit for managing transactions in SQL databases and (2) eventual consistency in managing any distributed application.

Laigner et al. (2021). attempted to bridge the gap of a lack of thorough investigation of the state of the practice and the major challenges faced by microservice architecture practitioners regarding data management. The authors identified three main reasons why a microservice architecture should be adopted regarding data management: (1) functional partitioning is used to support scalability and high data availability, (2) decentralised data management provides the ability to manage data store schemas for each microservice independently, and (3) even driven architecture allows for a reactive application to be built.

Database and deployment patterns were investigated by Laigner (2021). 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.

Brown et al. (2016) researched the implementation patterns for microservice architectures and proposed a pattern language. 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 to 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 Ntentos et al. (2020) in the article Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices. According to the authors, three things have to be considered while choosing data storage: reliability quality, scalability quality, and adherence to best practices of microservice architecture. The most recommended option is the database per service pattern, and the second option is to use a shared database, but it negatively affects the loose coupling quality.

Messina et al. (2016) proposed and tested a simplified database pattern for microservice architecture where a database is a separate microservice itself. 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 have no third-party dependencies, database microservices encapsulate 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.

Villaca et al. (2020) evaluated the use of a multistore database canonical data model in a microservice architecture. 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 utilisation 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 to data persistence in microservice architecture was presented by Viennot et al. (2015) in the paper Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications. 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 database. Synapse transparently synchronises shared data subsets between different databases in real-time. Synchronisation 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.

To sum up, it can be stated that there are no criteria, based on which SQL or NoSQL could be chosen as a database for microservice. Instead, there are recommendations when SQL or NoSQL could be a better option. For example, 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 transaction consistency is considered 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 Wiese (2015), multi-model polyglot persistence is recommended only if a limited set of data models is required to be accessed.

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 for 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. The proposed approach of monolith database migration into multi-model polyglot persistence based on microservice architecture is provided in Chapter 2.1.4, and its evaluation is provided in the Fourth Chapter.

1.6. Conclusions of the First Chapter and Formulation of the Tasks of the Dissertation

The first chapter of the dissertation provides an overview of microservice architecture and migration from monolith architecture to microservice architecture. The following conclusions have been drawn:

1. Microservice architecture has many advantages over monolithic architecture and has become a standard by default for modern cloud-based software systems in most enterprises. Many enterprises have started modernising their legacy monolithic applications by decomposing to microservices to remain competitive. Migrating from a monolithic to a microservices architecture poses challenges such as defining appropriate service boundaries, handling data management shifts from a unified to a distributed model and managing complex inter-service communication. The process involves a more intricate deployment and monitoring system, introduces testing and service coordination complexities, and necessitates an understanding of distributed systems. Furthermore, the transition may introduce network latency, potentially affecting performance. As microservices architecture is so complex and a relatively new architectural style, no widely approved way of conducting a migration from monolithic architecture to microservice architecture exists.
2. Three main challenges of migration to microservice architecture have been identified: microservice extraction from legacy monolith code bases, communication establishment between decomposed microservices, and data management adaptation to microservice architecture. While microservice extraction from legacy monolith code bases has already been explored by scientists and software engineers, there is very little research communication between microservices and data management.
3. The number of extracted microservices and the size of each microservice depend on the chosen code decomposition method. Code-based and storage-based methods allow for the identification of different technical functions and group code and storage components based on them. Business domain-based methods allow the decommissioning of applications into microservices based on identified business domains. Code-based and storage-based methods provide higher granularity.
4. The code quality of legacy monolithic applications has a great impact on the migration process. The better quality is, the less effort is needed to migrate from monolithic to microservice architecture.
5. Each microservice can be different in a variety of aspects, and no one database could potentially satisfy all the needs, which naturally leads to the use of polyglot persistence as a microservice data store.

Based on the conclusions, the following tasks are formulated to achieve the goal of the dissertation:

1. To investigate communication technologies for microservices and determine particular cases for their use.

2. To propose and evaluate the approach of monolith database migration into multi-model polyglot persistence based on microservice architecture.
3. To propose a new approach to migration from legacy monolith application to microservice architecture, which will combine code decomposition, establishing communication between microservices and data management areas.

2 Approach to Migrating a Legacy Monolithic Application in Microservice Architecture

This chapter proposes an approach that allows migrating existing legacy monolith applications into a microservice architecture. Migration from a monolithic architecture to a microservice architecture is a complex challenge, which consists of many different issues, such as microservice identification, code decomposition, commination establishment between decomposed microservices, independent deployment, data storage adaptation, etc. Unlike other migration approaches, the proposed migration approach consists of three parts: code decomposition, communication establishment and database migration. The primary focus of most of the other research is microservice identification within monolith applications and source code decomposition into microservices. All of the existing migration methods provide very little or no recommendations on how to adapt data storage to a microservice architecture and how to establish the connection between microservices during the migration from a monolith to a microservice architecture.

Two publications were published on the topic of this chapter (Kazanavicius, Mazeika, Kalibatiene et al., 2022; Kazanavicius, Mazeika et al., 2023).

2.1. Proposed Migration Approach

The main steps of the proposed approach for the migration from legacy monolith application to microservice architecture are shown in Fig. 2.1. It consists of five main steps, divided into several sub-steps: *Step 1 – Analysis of an existing monolith application; Step 2 – Monolith code decomposition into microservices; Step 3 – Communication establishment between microservices; Step 4 – Database adaptation to microservice architecture; Step 5 – Release and deployment*. A detailed explanation of each step and its sub-steps is provided next.

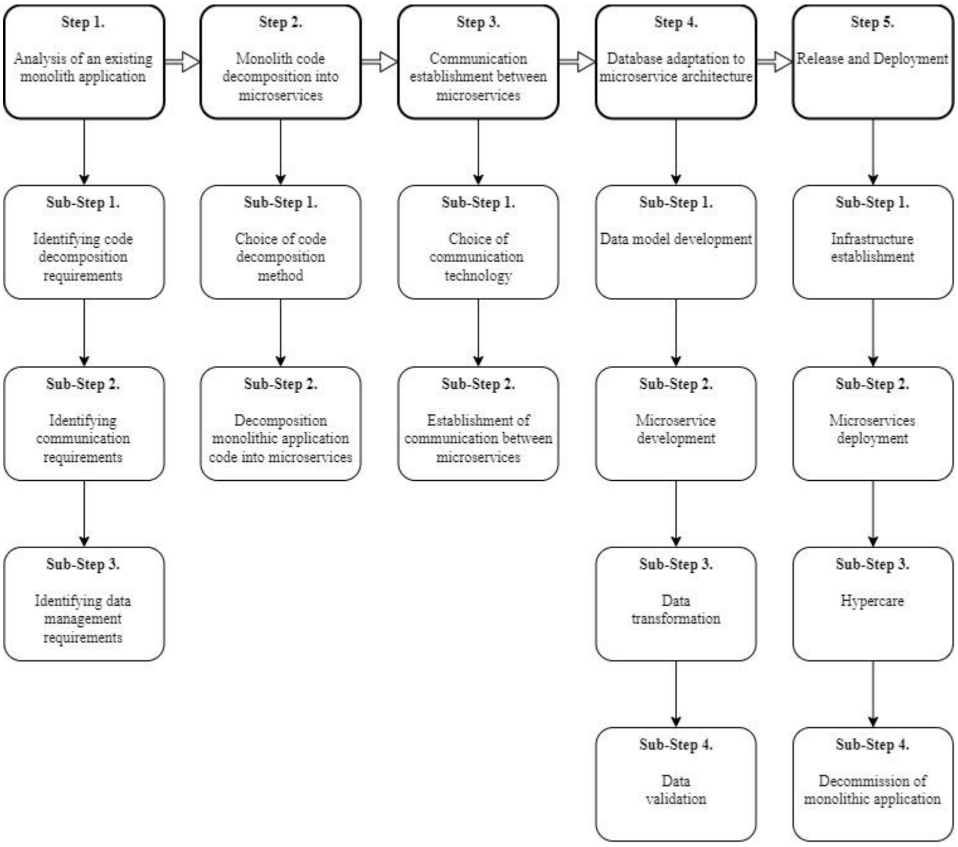

Step 1.
Analysis of an existing monolith application
Step 2.
Monolith code decomposition into microservices
Step 3.
Communication establishment between microservices
Step 4.
Database adaptation to microservice architecture
Step 5.
Release and Deployment
Sub-Step 1.
Identifying code decomposition requirements
Sub-Step 2.
Identifying communication requirements
Sub-Step 3.
Identifying data management requirements
Sub-Step 1.
Choice of code decomposition method
Sub-Step 2.
Decomposition monolithic application code into microservices
Sub-Step 1.
Choice of communication technology
Sub-Step 2.
Establishment of communication between microservices
Sub-Step 1.
Data model development
Sub-Step 2.
Microservice development
Sub-Step 3.
Data transformation
Sub-Step 4.
Data validation
Sub-Step 1.
Infrastructure establishment
Sub-Step 2.
Microservices deployment
Sub-Step 3.
Hypercare
Sub-Step 4.
Decommission of monolithic application

Fig. 2.1. Proposed approach of migration from legacy monolith application into a microservice architecture

2.1.1. Analysis of an Existing Monolith Application

Step 1 aims to analyse existing legacy monolith applications and identify functional and non-functional requirements for the next steps. Three types of requirements must be gathered: for monolith code decomposition, communication establishment between microservices, and database adaptation to microservice architecture.

Gathering monolithic code decompression requirements requires answering the following question: *What is the microservice's responsibility?* There are two types of responsibilities: business domain and technical function. Depending on the microservice's responsibility, Step 2 will be to choose a decomposition method. Microservices based on technical function provide higher granularity. Another important aspect which has to be identified is *code quality*. The better the code quality, the easier and faster it is to extract functions from it. If the code quality is very low, it may not be possible to use code-based decomposition methods.

To help choose the most appropriate communication technology for microservices, the author has provided the list of criteria: *performance*, *message size*, *memory size*, and *storage size*. Performance requirements, such as latency and throughput, should be provided. Message size, message complexity, and network load have the biggest impact on latency and throughput. Hence, these metrics have to be specified at the beginning to choose the most appropriate communication technology. Message size and network load should also be used to evaluate the impact on network bandwidth. The larger the message or the higher the network load, the greater its impact on network bandwidth. In case there is a network limitation, the size of the message and network load have to be considered. While evaluating microservice memory and storage consumptions, other environmental limitations, such as memory or storage size, have to be considered as well. A need for horizontal scalability could also be evaluated, as some communication technologies have this feature built-in, while others require additional tools and effort.

The database requirements consist of *functional requirements* and *data models* of existing legacy monolith applications. Domain experts and IT experts have to work together to identify all functional requirements and build the most optimum data model. A business analysis must be conducted to identify business processes and their data models. Understanding business logic is crucial to list the essential business rules. Once business rules are clear, technical analysis related to business rules has to be conducted to identify functional requirements for the database. Finally, a data model of existing legacy monolith applications has to be identified. To achieve the optimal data model results, a top-down approach is recommended to use instead of a bottom-up one.

2.1.2. Monolith Code Decomposition into Microservices

During Step 2, a code decomposition method has to be chosen and based on it; a legacy monolith application has to be decomposed into microservices (Fig. 2.2). The proposed approach provides three decomposition methods to choose from: *Code based* – application decomposition should be implemented based on code items like class. Application functions should be identified, and all code items should then be assigned to one of these functions. *Business domain-based* – applications should be divided into business domains, and each business domain should have a separate microservice. *Storage-based* – all the code related to specific storage items like databases or database tables should be placed in one microservice. More details about methods and their evaluations are provided in the first chapter.

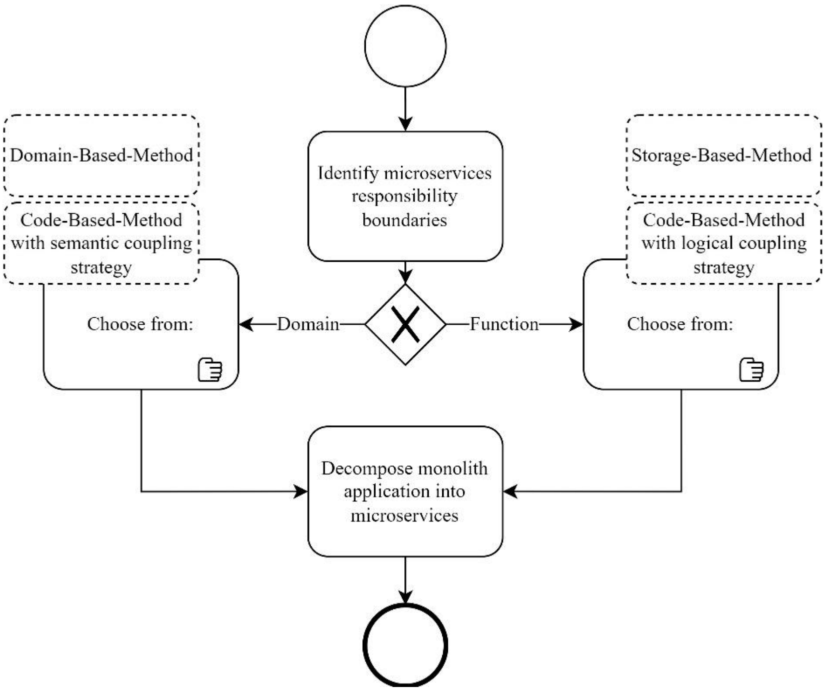

Domain-Based-Method
Code-Based-Method with semantic coupling strategy
Identify microservices responsibility boundaries
Storage-Based-Method
Code-Based-Method with logical coupling strategy
Choose from:
Domain
Function
Choose from:
Decompose monolith application into microservices

Fig. 2.2. Monolith code decomposition into microservices

One of the main principles of microservice architecture is that it should have only one responsibility. There are two types of responsibilities: business domain and business or technical function. Business domain responsibility is bigger than functional responsibility because it contains at least one function, and usually, it contains much more than one. Split by functions, microservices are smaller and have been called serverless.

If an organisation decides to decompose legacy monolith applications into microservices based on business domains, then it is recommended to choose the *business-domain-based method* or *the code-based method with a semantic coupling strategy*. If the decision is to decompose legacy monolith applications into microservices based on functions, then *the storage-based method* or *code-based method with a logical coupling strategy* could be used. The real difference in microservice size depends on how much the business domain contains functions. The more functions the business domain will have, the bigger the microservices candidate will be extracted using the business-domain-based method or code-based method with a semantic coupling strategy.

If the organisation aims to have automatic decomposition, then the *code-based method with a contributor coupling strategy* should be chosen as it has the potential to be implemented fully automatically. The monolith must be implemented using object-oriented programming language because the extraction model is based on classes such as the atomic unit of computation and the graph. The *code-based method with a semantic coupling strategy* could be implemented semi-automatically. In this case, business domains should be identified manually. How accurately the method will be able to identify the class relation to the business domain depends on the naming convention in the code. The *storage-based methods* and business-domain-based methods cannot be implemented automatically. *The storage-based method* requires manually identifying business subsystems and assigning database tables to one of the subsystems. The *business-domain-based method* requires two manual analyses to do.

Choosing the right method for microservice extraction from the legacy monolith application method is a hard task, which is crucial for successful migration. Each legacy monolithic application is unique and creates unique challenges. Technology stack, complexity, business object, team size or skills, etc. are the things which could be very different in each case.

No one best methodology for extracting microservices from the monolith exists. Each case is different, and the organisation should choose which method or combination of methods best suits its migration from monolith to microservices. Each organisation has its reasons and goals for migrating from the monolith to microservices. The chosen extraction methodology should help to achieve those goals. Selected methodology or combination of methodologies should be:

- Able to extract microservices by selected factors and expected size.
- Compatible with technological stack and database technologies used in monolith applications.

2.1.3. Communication Establishment between Microservices

The main goal of Step 3 is to choose communication technology and establish communication between microservices decomposed from the legacy monolith applications in Step 2. The proposed approach provides five communication technologies to choose from: (1) *HTTP Rest* – usually, HTTP/HTTPS protocols are used for synchronous communication. The main advantage of this communication is that the system is simple and easily implemented. Also, there is no intermediate component, such as a message broker. (2) *RabbitMQ* is open-source general-purpose broker-based asynchronous communication technology. RabbitMQ natively implements an Advanced Message Queuing Protocol. It was originally developed by Rabbit Technologies Ltd. (3) *Kafka* is an open-source distributed publish–subscribe messaging system. Instead of relying on a message queue, Kafka stores messages to the stream and allows consumers to pool. It was originally developed by the Apache Software Foundation. (4) *gRPC* is an open-source Remote Procedure Call (RPC) framework developed by Google. It enables the establishment of transparent communication between server and client applications in any environment. Before gRPC became open source, it was used as a single general-purpose RPC infrastructure to connect the large number of microservices running within and across Google data centres for over a decade. (5) *GraphQL* is a query language for APIs and a runtime for fulfilling those queries with existing data. GraphQL was developed internally by Facebook in 2012 and was published to the community in 2015. The key functionality of the GraphQL framework is a query language that allows clients to define the structure of the data required, and the same structure of the data is returned from the server. More details about technologies are provided in the first chapter, and their experimental evaluations are provided in the third chapter.

One of the most significant challenges during migration from legacy monolith applications into microservices is data communication management. How are in-process methods or function calls in monolith applications transformed into inter-process communication? The high complexity, variety of architectural aspects, technological stack, and business objects make every application different and create challenges during monolith application decomposition to microservices. The proposed approach provides criteria based on communication technology (Fig. 2.3).

If latency and throughput are the main criteria, then RabbitMQ and gRPC are the most suitable technologies. RabbitMQ showed the best results in RPC latency and throughput tests for small messages (up to 0.1 MB and a data model up to 100 properties), while gRPC showed the best results in RPC latency and throughput tests for big messages.

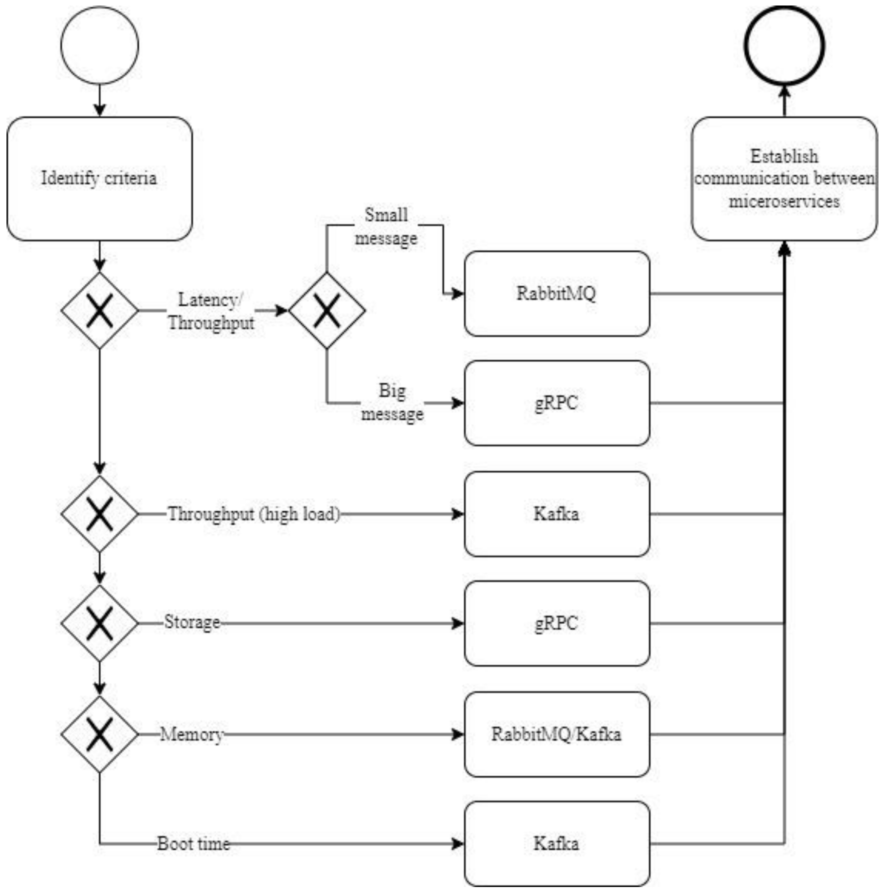


Fig. 2.3. Communication establishment between microservices

Kafka showed the best throughput results in the most loaded conditions: requested by more than 100 clients at the same time and processing 1,000,000 characters of messages. *However, the latency of RPC was high, more than one second*. HTTP Rest has the smallest request and response message size. If message size is an important criterion when choosing communication technology, then HTTP Rest is a recommended technology. On the other hand, gRPC has the smallest payload as it uses binary serialisation. Theoretically, at some point of complexity, for complex data models with many properties, gRPC request and response message size should become smaller than HTTP Rest. Deeper research is needed to determine the exact complexity threshold. The gRPC library is using the least amount of storage. If microservices are running in an environment with limited storage, then gRPC must be used. RabbitMQ and Kafka consume the smallest amount of memory. Therefore, if memory size is one of the essential criteria, then

RabbitMQ and Kafka must be used for implementation. Microservice implemented using Kafka library boots up the fastest.

If horizontal scalability is an important aspect, Kafka and RabbitMQ are the best candidates as they have built-in cluster functionality. It must be noted that other technologies can be scaled horizontally as well, but it requires additional tools and effort. HTTP Rest and RabbitMQ are prevalent communication technologies, and many different libraries exist in the market to choose from, while GraphQL and gRPC are relatively new and rapidly growing communication technologies with fewer libraries to choose from. Synchronous communication style communication technologies gRPC, HTTP Rest, and GraphQL do not require any additional components to communicate, while asynchronous communication technologies RabbitMQ and Kafka require service as an interim communication layer. Hence, additional components increase solution complexity and maintenance costs. On the other hand, if a solution contains many microservices and scalability is a challenge, RabbitMQ and Kafka as an interim layer can provide centralised communication routing functionality.

2.1.4. Database Adaptation to Microservice Architecture

During Step 4, the existing legacy monolith application database has to be adapted to microservice architecture. The purpose of the proposed approach is shown in Fig. 2.4.

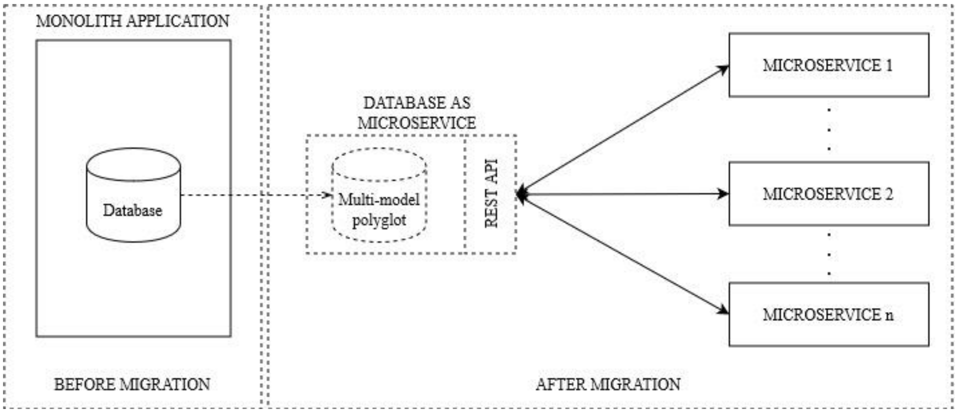


Fig. 2.4. Purpose of the proposed database migration approach

The approach can extract a database from a monolith application and transform it into a multi-mode 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 the benefits of microservices, such as agility and scalability, to be used better. The encapsulation of a database into a microservice reduces the complexity and increases the performance. After migration, the data are accessible not only to an existing monolith application but also to any microservice within an ecosystem.

The proposed approach of migration from a monolith database to multi-model polyglot persistence based on microservice architecture is shown in Fig. 2.5. It consists of four steps, and each step is divided into sub-steps. A detailed explanation of each step and its sub-steps are provided in the next chapters.

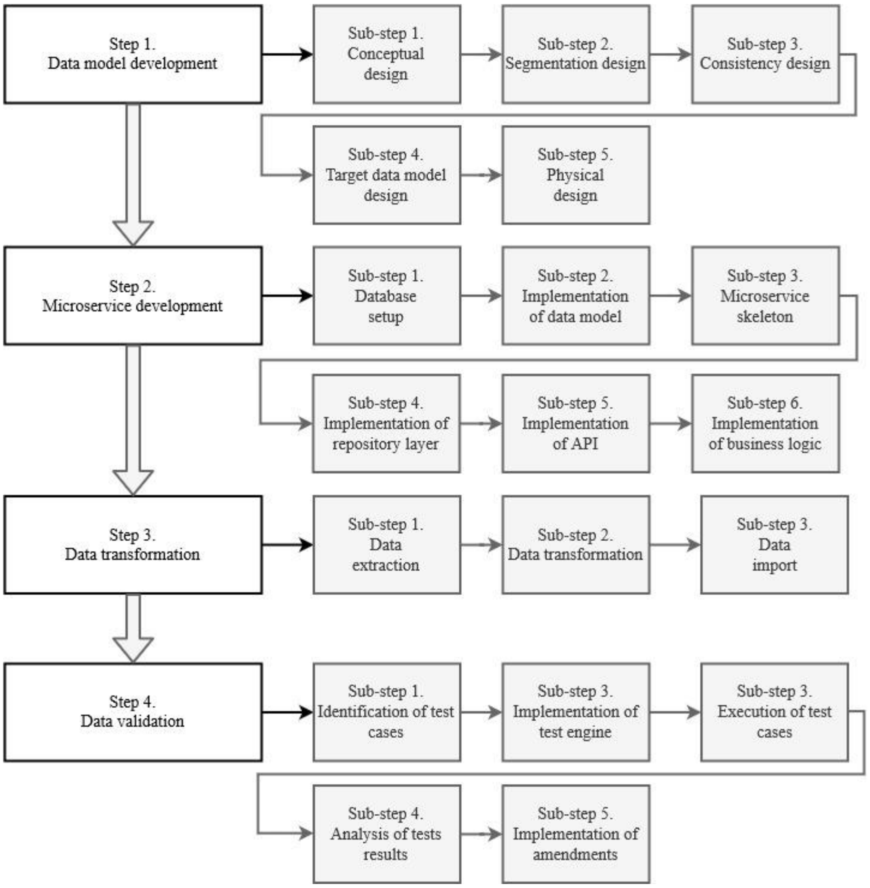

Step 1.
Data model development
Sub-step 1. Conceptual design
Sub-step 2. Segmentation design
Sub-step 3. Consistency design
Sub-step 4. Target data model design
Sub-step 5. Physical design
Step 2.
Microservice development
Sub-step 1. Database setup
Sub-step 2. Implementation of data model
Sub-step 3. Microservice skeleton
Sub-step 4. Implementation of repository layer
Sub-step 5. Implementation of API
Sub-step 6. Implementation of business logic
Step 3.
Data transformation
Sub-step 1. Data extraction
Sub-step 2. Data transformation
Sub-step 3. Data import
Step 4.
Data validation
Sub-step 1. Identification of test cases
Sub-step 3. Implementation of test engine
Sub-step 3. Execution of test cases
Sub-step 4. Analysis of tests results
Sub-step 5. Implementation of amendments

Fig. 2.5. Proposed database migration approach

During Step 1 (Fig. 2.6), 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 consists of five sub-steps:

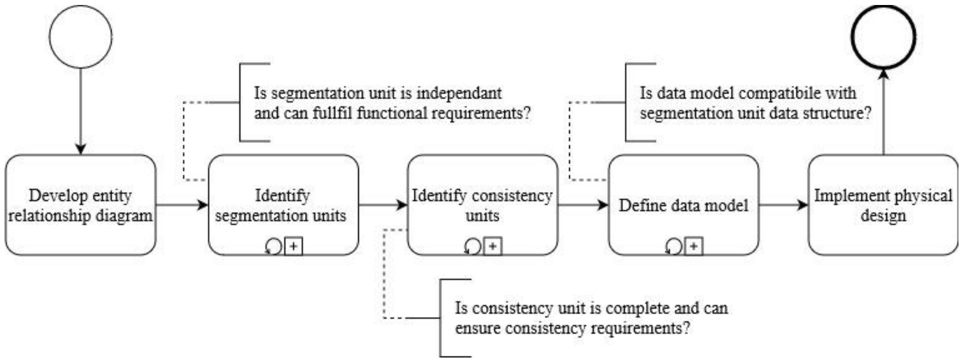

Is segmentation unit is independant and can fullfil functional requirements?
Is data model compatibile with segmentation unit data structure?
Develop entity relationship diagram
Identify segmentation units
Identify consistency units
Define data model
Implement physical design
Is consistency unit is complete and can ensure consistency requirements?

Fig. 2.6. Data model development

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. *Consistency design* identifies consistency units to allow data fragmentation and horizontal scalability.
4. *Target data model design* chooses the best data structure for each identified segmentation unit from different data structures supported by multi-model polyglot persistence.
5. *The physical design* implements the built target data model into a multi-model polyglot persistence database. As each database is different, this sub-step aims to implement all technical peculiarities needed to support the developed target data model in the database.

The main goals of Step 2 are to set up the multi-model database and encapsulate it into the microservice. This allows for the implementation of the database as a service pattern, where a database is a microservice itself.

Sub-step 1 is to install a multi-model polyglot database and set up technical peculiarities, such as creating a 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 – the first approach where models and data structures are defined in an application.

The purpose of Sub-step 3 is to create a microservice skeleton that can 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; e.g., it could be an OpenShift project in a private cloud. The number of active containers and scalability settings has 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/CD) pipeline has to be set up to automate build, test, and deploy activities and ensure security so 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 centralised logging solutions. ELK Stack could be an example of a good logging solution.

The repository layer has to be built to provide microservice accessibility to the database in Sub-step 4. All actions needed to establish a connection between a database and a microservice have to be executed first, e.g., firewall rules, service account access rights, connection string, etc. The next step is the implementation of a repository layer. The code that can 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 5, the 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 a customer*, *viewing a customer*, *updating a customer*, or *deleting a customer*. All four methods have to be created in the customer's controller. The authentication and authorisation functionality has to be implemented to fulfil the security requirements and manage the accessibility to different methods.

The last sub-step aims to implement the business logic layer, which has to connect the API layer and the repositories layer. Because the 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 1 and connects the API and repository layers.

An example of one possible implementation is presented in the sequence diagram below (Fig. 2.7). The 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 general customer information, such as name, surname, address, etc. *GetCustomerHistory* obtains the customer's payment history.

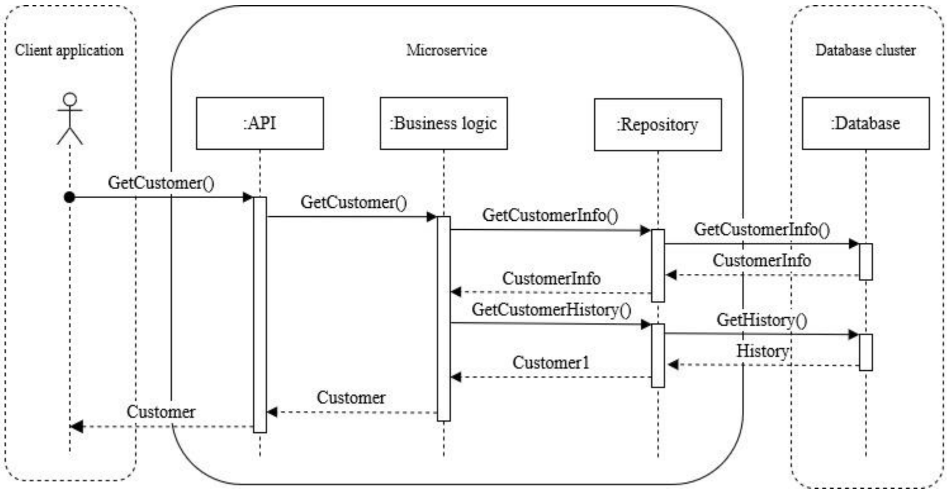


Fig. 2.7. Example of the function *GetCustomer* implementation within the microservice layers

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 API layer is responsible for data exposure to clients via the API interface and encapsulates all technical implementation peculiarities, such as connection establishment, authorisation, authentication, etc. The business logic layer is responsible for building a consistent domain data model.

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. This step aims to create an application that can execute data transformation between databases. The proposed data transformation process is shown in Fig. 2.8.

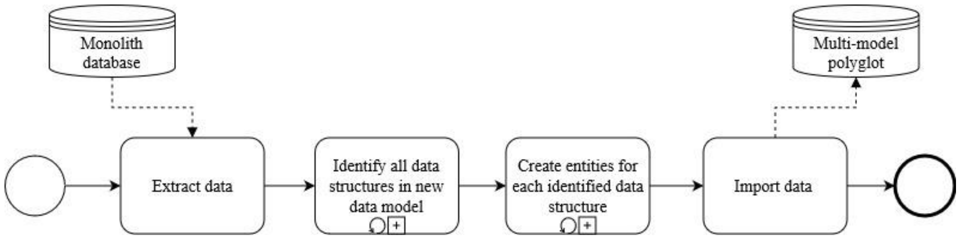

Monolith database
Multi-model polyglot
Extract data
Identify all data structures in new data model
Create entities for each identified data structure
Import data

Fig. 2.8. Proposed data transformation process

Sub-step 1 is to extract everything needed to transform the data from a monolith database. A code that can read data from a monolith database and transform it into data models that represent the used data structure has to be written. The author recommends 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 is shown in Fig. 2.9. The simplified example of the data model of multi-model polyglot persistence is shown in Fig. 2.10. 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 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 Step 3, 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.

```
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; }
}
```

Fig. 2.9. Example of the simplified repository and model implementations

```
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,
            }
        };
    }
```

Fig. 2.10. Example of simplified data model mapping

It is a combination of two data models used in monolith applications. The *MonolithModel1* and *MonolithModel2* models represent two data tables in the monolith database, and *PolyglotModel* represents a document with the embedded subdocument *PolyglotChildModel*. Even though the given example looks straightforward, in practice, the mapping logic can be more complicated: the data model for the polyglot can be a combination of dozens of data tables, and 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 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);
        }
    }
}
```

Fig. 2.11. Example of the simplified record creation class

The next action in Sub-step 2 is to create all records for polyglot persistence based on records in a monolith database. In examples defined in Figs. 2.9 and 2.10, 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 record creation is shown in Fig. 2.11. 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 Sub-step 2 into a multi-model polyglot database installed in the third step. The author suggests reusing the repository layer created in the microservice.

Even though the data transformation process could be implemented in different ways, the author recommends building a separate application for this purpose. This would allow for the process to be repeated as many times as needed if 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.

The purpose of Step 4 is to create automatic data validation. Transformed data have to be validated before it is released for production. In Sub-step 1, test cases have to be created based on the functional requirements and data in the mainframe monolith database. Sub-step 2 is to create a test engine that has to be able to execute the 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 (Fig. 2.12). The step is finished only when all of the test cases are passed.

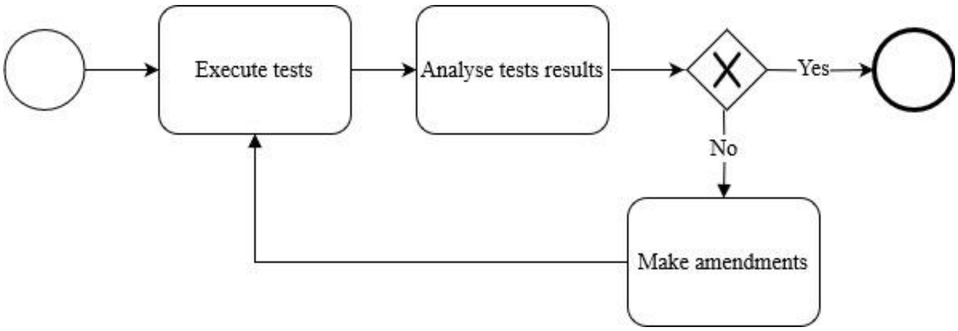


Fig. 2.12. Test case execution

For example, the functional requirements for the data records of *PolyglotModel* defined in Fig. 2.10 are *read*, *create*, *update*, and *delete*. Four test cases have to be created to validate the data integrity and persistence, and one test case for one functional requirement. The first functional requirement is the possibility to read the data. In this example, it is possible to read data records by *Polyglot-*

Model. Two main criteria have to be verified. First, in each data record of *PolyglotModel*, all fields have to be mapped correctly, and the data must 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. Fig. 2.13 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 that 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 3, 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"
   }
};
```

Fig. 2.13. Example of the data records in the monolith database and multi-model polyglot persistence

Three more test cases have to be created to validate functional requirements: *create*, *update* and *delete*. The test case for *creation* 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 deleted.

2.1.5. Release and Deployment

The last step aims to release and deploy extracted microservices and adapted databases. It includes all technical peculiarities needed to deploy and run microservices and databases.

First, an execution environment has to be chosen and prepared for extracted microservices. The two most common options are virtual machines and containers. While virtual machines virtualise hardware and OS, containers virtualise only OS. The possibility of running multiple containers on a single operating system makes containers advantageous in terms of scalability, lower cost, efficient resource usage, portability, etc. The biggest advantages of virtual machines are that they have harder security boundaries and more resources. There is the possibility of running a few microservices in a virtual machine; however, it compromises the single biggest advantage of breaking down a monolithic application into small, easily executable microservices. Even though it is possible to run microservices in virtual machines, the author strongly recommends the use of containers as they better utilise microservice architecture advantages.

A CI/CD pipeline should be set up for each microservice to make them independent. The philosophy of microservices states that there should never be a long release queue where every team has to get in line. There should be no dependencies, and the team that builds microservice “X” should be able to release it at any time without waiting for any changes in microservice “Y”. To achieve a high release velocity, the release pipeline has to be automated as much as possible. Each organisation should decide on a strategy on how to do it and choose tools for it: source control – where and how should be stored and maintained source code, build tool – how microservice should be built, tests tool – how tests should be run, and deploy tool – how microservice should be deployed.

Monitoring and logging are other important aspects to be considered while building infrastructure for microservices. Microservices are distributed applications, and the flow goes through multiple processes. It is difficult to get a holistic view of the entire application and its flow. To do it efficiently, monitoring and logging services should be centralised and have a strong visualiser.

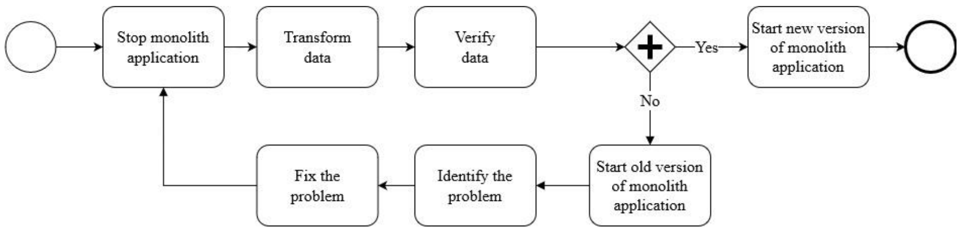

Stop monolith application
Transform data
Verify data
Yes
Start new version of monolith application
No
Start old version of monolith application
Identify the problem
Fix the problem

Fig. 2.14. Deployment of microservices

Once the infrastructure is established, all microservices can be deployed into the production environment (Fig. 2.14). At first, the monolith application has to be stopped, and data transformation has to be executed. The precondition for the deployment of all microservices is successful data transformation.

Sub-step 3 is hyper care, during which domain and IT experts have to give hyper attention to newly released software and fix any last errors if they appear. The last sub-step is decommissioning an unused monolith application and database.

2.2. Conclusions of the Second Chapter

The second chapter of the dissertation proposes an approach that allows migrating the existing legacy monolith applications into a microservice architecture. The following conclusions have been drawn:

1. To bridge the existing gaps in communication and database management, a novel approach is proposed for migration from legacy monolithic software to microservice architecture. It consists of five steps: *Step 1 – Analysis of an existing monolith application, Step 2 – Monolith code decomposition into microservices, Step 3 – Communication establishment between microservices, Step 4 – Database adaptation to microservice architecture,* and *Step 5 – Release and Deployment*.
2. The proposed novel approach allows conducting database migration from monolith architecture into a microservice architecture by transforming the existing data model into multi-model polyglot persistence that is embedded in a microservice and exposed via an API.
3. Novel evaluation criteria are proposed, according to which code decomposition methods and communication technologies are selected, considering their advantages and disadvantages.

3

Investigation of Microservice Communication while Decomposing Monoliths

One of the biggest challenges while migrating from a monolith architecture to a microservice architecture is to define a proper communication technology. In monolith applications, communication between components is performed using the in-process method or function calls, while different communication methods have to be established to achieve the same functionality in a microservice architecture. A microservices-based application is a distributed system running on multiple processes or services. Therefore, microservices must interact using inter-process communication technologies.

This chapter provides an analysis of how proper communication between decomposed microservices could be established. A set of criteria, which is important while decomposing monoliths to microservices, was identified. The benefits and drawbacks of communication technologies and the impact on communication between microservices were evaluated based on these criteria. Five technologies were chosen for analysis, e.g., HTTP Representational State (Rest) API, RabbitMQ, Kafka, gRPC, and GraphQL. Rest API represents an asynchronous communication style and has become a de facto standard synchronous communication technology. RabbitMQ and Kafka represent asynchronous communication based on a message broker. GraphQL and gRPC have been selected for the investigation because of their rapidly growing popularity. GraphQL provides the functionality of client-side applications to query databases at server-side applications, while gRPC is a technology that implements remote procedure call (RPC) API. It uses HTTP 2.0 as its underlying transport protocol and is provided as a data structure. Various criteria were considered while analysing selected communication technologies, including influence on microservice topology, the performance of remote procedure calls, message size, memory consumption, storage usage, boot time, and availability of the corresponding libraries. The main contribution of this work is a unique set of criteria used to compare five communication technologies and evaluate their advantages and disadvantages in the context of monolith decomposition to microservices. The key findings identified during this research are provided as a guideline for the researchers and industry that can help to speed up legacy monolith decomposition to microservices and make this complex procedure more obvious.

One publication was published on the topic of this chapter (Kazanavicius, Mazeika et al., 2023).

3.1. Evaluation of Microservice Communication

A set of five microservices was created and connected in a line topology to evaluate and compare communication technologies (Fig. 3.1). The RPC technique was used for communication between microservices. Only pure server and client functionality were implemented in each microservice; the server component exposes API, and the client component is used to execute RPC. The experiment aimed to evaluate and compare communication-based on the remote procedure call (RPC). RPC technique was chosen because it supports the same functionality as a function call and in-process-based communication.

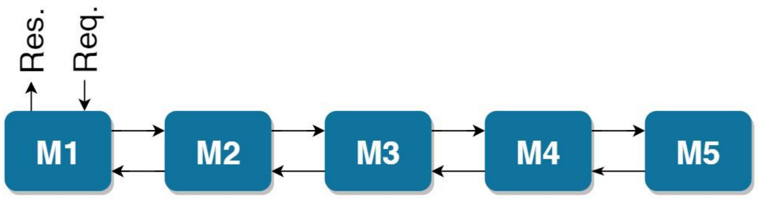


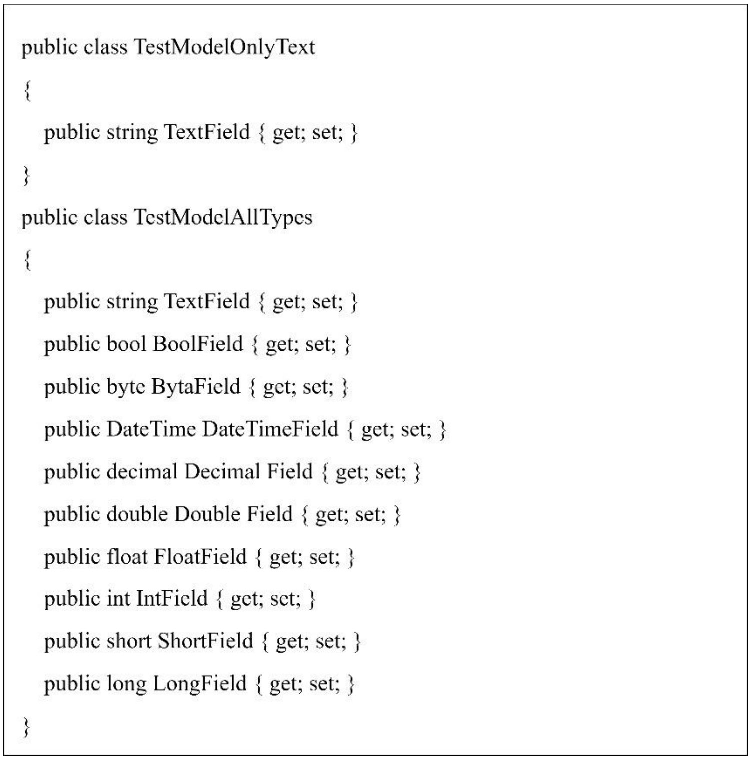
Fig. 3.1. Topology of microservices used for the experiment. Where: Req. is a request, Res. is a response, and Mi is a microservice

The full flow of message processing in the conducted experiment is defined as follows:

$$t = \left(\sum_{i=1}^{n=4} M_i \to M_{i+1}\right) + \left(\sum_{i=0}^{n=3} M_{5-i} \to M_{5-i-1}\right), \tag{3.1}$$

where: t is the time used to process the message, Mi is microservice with index *i*, and arrow (→) is request/response operation. Different size and complexity messages were sent to evaluate and compare the impact of message size, message complexity and request load on the latency and throughput of each technology. The time duration between requests sent from M1 to M5 and the response received from M5 to M1 was measured and was used to calculate latency and throughput.

Different data models were used (Fig. 3.2) for messages to measure the impact of message size and complexity on latency and throughput. The *TestModelOnlyText* data model was used to measure the impact on message size; the *TextField* value was set to 10, 1,000, 100,000, and 1,000,000 characters. The *TestModelAllTypes* data model was used to measure the impact on message complexity, especially on serialisation. Messages with 10, 100, 1,000 and 10,000 properties were used.



```
public class TestModelOnlyText
{
   public string TextField { get; set; }
}
public class TestModelAllTypes
{
   public string TextField { get; set; }
   public bool BoolField { get; set; }
   public byte BytaField { get; set; }
   public DateTime DateTimeField { get; set; }
   public decimal Decimal Field { get; set; }
   public double Double Field { get; set; }
   public float FloatField { get; set; }
   public int IntField { get; set; }
   public short ShortField { get; set; }
   public long LongField { get; set; }
}
```

Fig. 3.2. Data models used in the experiment

The latency was measured by processing different sizes and complex messages while requesting using one client. The throughput was measured by processing the same messages as it was processed in latency tests but with an increased request load. During the experiment, the request load started with ten clients and was constantly increased by ten clients every 30 seconds until it reached 200 clients.

3.2. Evaluation Criteria of Microservice Communication

This chapter provides information about criteria that were considered while analysing different communication technologies. Previous research performed by different authors was mainly focused on performance evaluations and comparisons. To cover more communication aspects that can potentially be a challenge during legacy monolith application decomposition to microservices, a set of new criteria was introduced. These criteria were chosen to compare each communication technology in the context of communication between microservices decomposed from monolith applications.

- *Performance*: communication technology performance is measured and analysed by latency and throughput. Latency was measured by time in milliseconds since the request was sent till the response was received. Throughput was measured by the number of successful requests per second (RPS). The successful request was considered if a response was received within one second.
- *Message size*: to determine the potential technology impact on network load request and response, message size in bytes was measured during the experiment.
- *Memory size*: to evaluate how much memory is needed to run an application with each communication technology, application memory usage in bytes was measured.
- *Storage size*: to evaluate how much storage is needed to store an application with each communication technology, storage usage in bytes was measured.
- *Boot time*: application boot time in seconds was measured to determine how much time is needed to start the application.
- *Architecture*: to highlight the specific impact of each technology regarding application architecture.

- *Topology*: technology impact on the topology of microservices. More details about the topology used in the experiment are provided in Chapter 3.4.
- *Used applications and libraries*: to analyse the availability of the particular library.

3.3. Topologies Used in Microservice Communication Evaluation

Three different topologies of microservices were chosen to analyse how communication technology influences topology criteria defined in the previous chapter (Fig. 3.3).

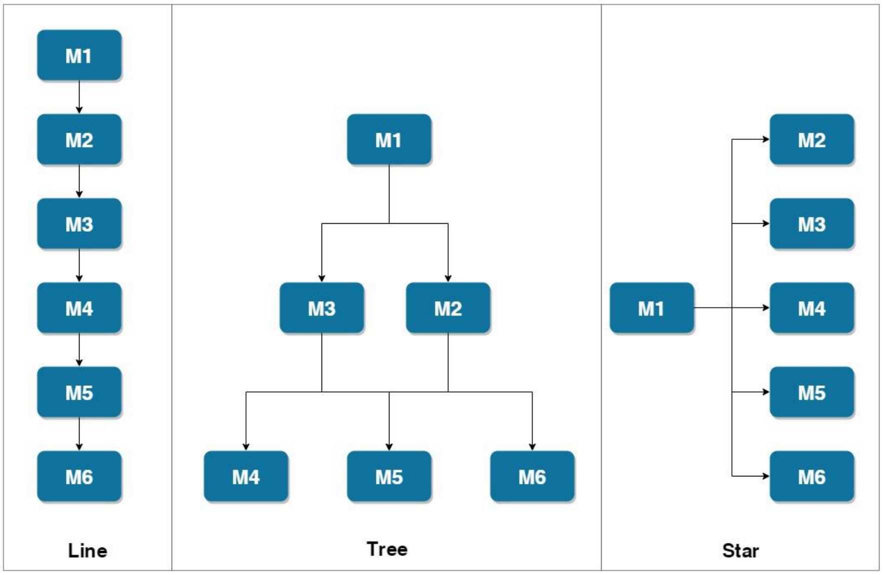


Fig. 3.3. Topologies used in the experiment

Linear (single receiver) topology – request processing flow has only one way in, and each microservice is involved in request processing. *Tree type topology* – request processing flow has a few ways. Middleware microservices work as gateways. *Star-type topology* (multiple receivers) – the first microservice works as a gateway and routes requests to a specific microservice. Those topologies were chosen because each of them represents a different way in which data can be processed, and communication between microservices can be established.

3.4. Tools Used in Microservice Communication Evaluation

All microservices were written using C Sharp and .Net Core. All coding and testing were done using Microsoft Visual Studio 2022 IDE. All libraries used in the research were downloaded from the NuGet gallery. Latency tests were conducted using the BenchmarkDotNet library. Throughput tests were executed by using the NBomber library. Network data was analysed using the Wireshark application.

All experiments were performed on a computer with the following specifications: CPU – Core i7 9850H, memory – 30 GB RAM, storage – 512 GB SSD, and OS – Windows 10 Enterprise (20H2). All applications were run on a computer, and no external devices or networks were used.

The experiment can be reproduced on a computer with Visual Studio 2022 IDE, RabbitMQ (3.10.0 version) and Kafka (3.2.0). The source code used in the experiment and experimental results are freely accessible and can be found at the following link: https://bitbucket.org/justas_kazanavicius/communicationexperiment.

3.5. Evaluation Results of the Microservice Communication Experiment

This chapter provides results obtained during the evaluation of five communication technologies: HTTP (Rest API), RabbitMQ, Kafka, gRPC, and GraphQL. Deeper discussions on results are provided in Chapter 3.6. Each section on technology is divided into six sub-chapters to provide more details in terms of experiment results:

- *Latency results:* Latency evaluation results are based on message size and complexity.
- *Throughput results:* Throughput evaluation results are based on message size and complexity.
- *Results of other metrics:* Request/Response size, Microservice application size, Memory usage size, Boot time.
- *Architecture:* technology and libraries impact the architecture.
- *Topology:* technology and libraries impact the topology.
- *Libraries*: a list of libraries that were used in the experiment to establish a connection between microservices via particular technology.

3.5.1. Evaluation Results of Hypertext Transfer Protocol

Latency results: Results of the latency test are shown in Table 3.1. The best result, 7.265 ms, was achieved by processing 1,000-character messages. The worst result, 31.410 ms, was achieved by processing 1,000,000 characters messages.

Table 3.1. Latency test results for message processing with HTTP

Message size	Mean	Median	Min	Max
10 characters	7.527 ms	7.404 ms	5.801 ms	9.923 ms
1,000 characters	7.265 ms	7.149 ms	5.685 ms	9.459 ms
100,000 characters	11.745 ms	11.356 ms	9.543 ms	15.875 ms
1,000,000 characters	31.410 ms	30.563 ms	25.304 ms	44.212 ms
10 properties	8.236 ms	8.055 ms	6.465 ms	11.516 ms
100 properties	8.459 ms	8.408 ms	6.396 ms	10.940 ms
1,000 properties	9.826 ms	9.726 ms	7.567 ms	13.284 ms
10,000 properties	21.779 ms	21.096 ms	19.010 ms	26.546 ms

Throughput results: The throughput results of the load test are shown in Fig. 3.4. The best average results, 99.7 RPS, were achieved by processing ten properties messages. The worst average result, 4.7 RPS, was achieved by processing 1,000,000 character messages.

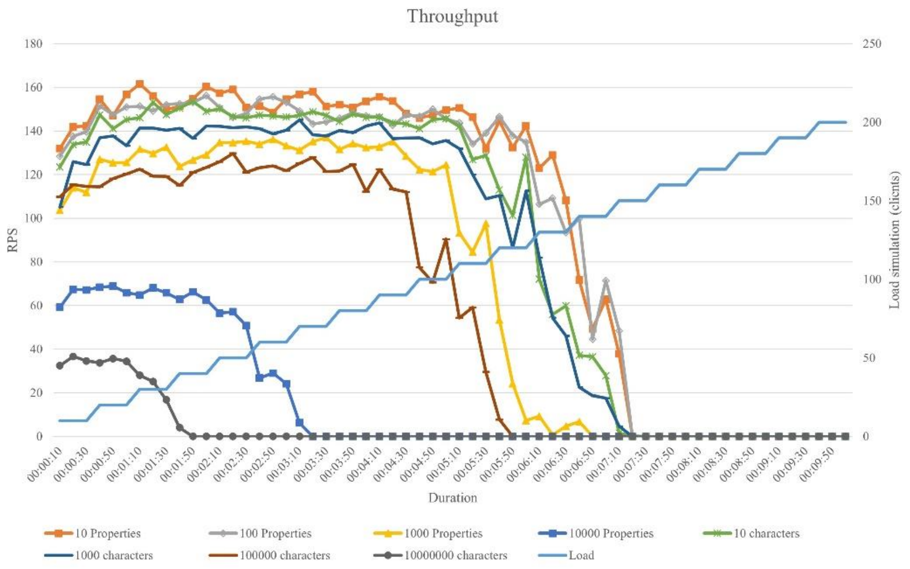


Fig. 3.4. Load test results for message processing with HTTP

Results of other metrics: Other results obtained during the experiment are presented in Table 3.2.

Table 3.2. Results of HTTP Rest experiment measurements

Metric	Result
Request/Response size	172 B/185 B (payload 26 B)
Microservice application size	4.71 MB (empty 159 KB)
Memory usage size	69 MB (empty 9 MB)
Boot time	3.1 seconds

Architecture: To communicate via Rest API, the microservice has to have at least three additional components: Rest API, Controller, and Rest Client (Fig. 3.5). Rest API component exposes the HTTP server and routes requests to the Controller component, which operates as a facade for business logic. Rest Client is needed to make requests to Rest APIs exposed by other microservices.

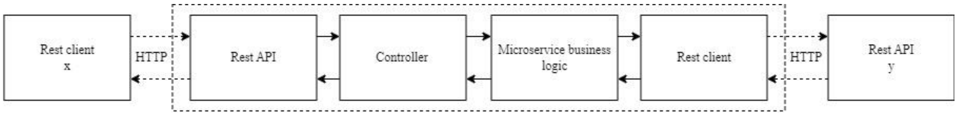


Fig. 3.5. Architecture of Rest API in microservice

Topology: Microservices M1–M5 have to know how to reach the next microservice (M1→M2, M2→M3, etc.) when a linear topology is used. Microservice M6 only exposes Rest API. The tree-type topology shows that microservices M1, M2, and M3 each have two dependencies (M1 should know the URLs of M2 and M3). M4, M5, and M6 only expose the Rest API. In the star-type topology, the M1 microservice has to know the URLs of all microservices.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via HTTP Rest technology is provided below:

- Microsoft.AspNetCore.App (Version 6.0.7)
- Microsoft.NETCore.App (Version 6.0.7)
- Swashbuckle.AspNetCore (Version 6.2.3)
- System.Net.Http.Json (Version 6.0.0)

3.5.2. Evaluation Results of RabbitMQ

Latency results: Results of the latency test are shown in Table 3.3. The best result, 2.976 ms, was achieved by processing 1,000-character messages. The worst result, 118.657 ms, was achieved by processing 1,000,000 characters messages.

Table 3.3. Latency test results for message processing with RabbitMQ

Message size	Mean	Median	Min	Max
10 characters	2.982 ms	2.946 ms	2.551 ms	3.491 ms
1,000 characters	2.976 ms	2.939 ms	2.721 ms	3.712 ms
100,000 characters	5.166 ms	5.023 ms	4.674 ms	6.360 ms
1,000,000 characters	118.657 ms	116.824 ms	73.740 ms	157.821 ms
10 properties	4.354 ms	4.265 ms	3.059 ms	6.605 ms
100 properties	3.197 ms	3.108 ms	2.843 ms	4.387 ms
1,000 properties	4.752 ms	4.670 ms	4.278 ms	5.875 ms
10,000 properties	20.310 ms	19.974 ms	19.529 ms	23.098 ms

Throughput results: Throughput results of the load test are shown in Fig. 3.6. The best average result, 231.5 RPS, was achieved by processing 10-character messages. The worst average result, 0.01 RPS, was achieved by processing 1,000,000 characters messages.

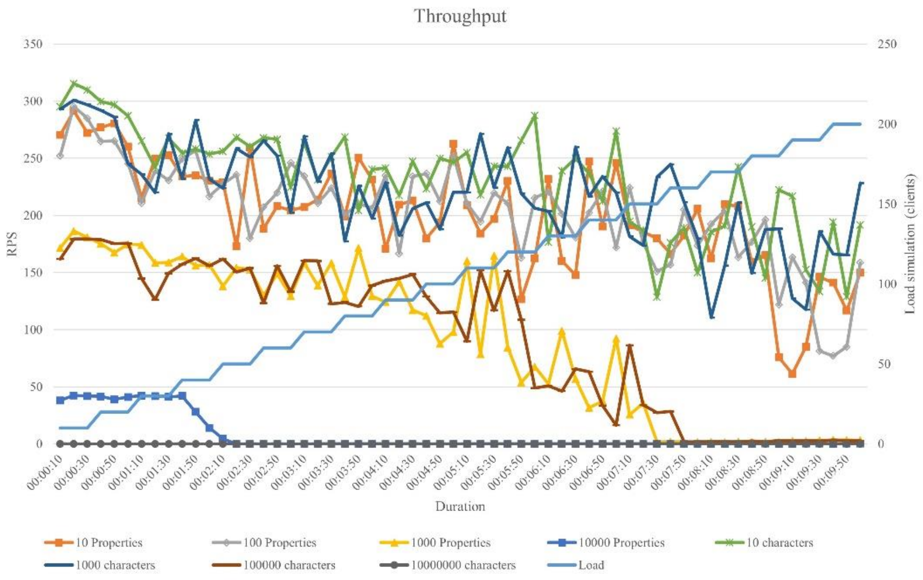


Fig. 3.6. Load test results for message processing with RabbitMQ

Results of other metrics: Other results obtained during the experiment are presented in Table 3.4.

Table 3.4. Results of RabbitMQ experiment measurements

Metric	Result
Request/Response size	206 B/225 B (payload 26 B)
Microservice application size	2.26 MB (empty 159 KB)
Memory usage size	23 MB (empty 9 MB)
Boot time	3.8 seconds

Architecture: To utilise RabbitMQ as RPC, microservices have to contain two additional components: a Rabbit server and a Rabbit client (Fig. 3.7). The Rabbit server consumes messages from queue x1 and routes them to business logic where messages are processed and moved to the Rabbit client to publish them to queue y1. After pushing messages to queue y1, the Rabbit client starts listening to queue y2 for a response. A message that is consumed from queue y2 goes from the Rabbit client through business logic to the Rabbit server, where it is published to queue x2.

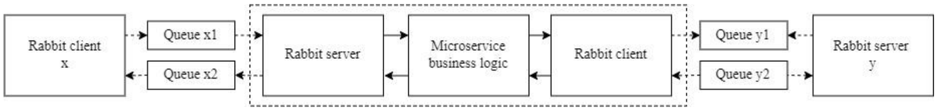


Fig. 3.7. Architecture of RabbitMQ in microservice

Topology: Similar to HTTP communication, the Rabbit server component is not needed for those microservices that are only used as clients, and the client component is not needed for those microservices that are only used as servers. The most significant difference using RabbitMQ is that there is no need for microservices to know about each other's endpoints, such as IP address or hostname. Instead of communicating directly with each other, microservices are communicating through RabbitMQ, which acts as a router. Clients are producers and produce messages to the RabbitMQ queue while servers are consumers and consume messages from the same RabbitMQ queue.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via RabbitMQ technology is provided below:

- Microsoft.NETCore.App (Version 6.0.7)
- RabbitMQ.Client (Version 6.3.0)
- Nito.AsyncEx (Version 5.1.2)

3.5.3. Evaluation Results of Kafka

Latency results: Results of the latency test are shown in Table 3.5. The best result, 7.191 ms, was achieved by processing 10-character messages. The worst result, 42.600 ms, was achieved by processing 1,000,000-character messages.

Table 3.5. Latency test results for message processing with Kafka

Message size	Mean	Median	Min	Max
10 characters	7.191 ms	7.130 ms	6.836 ms	8.023 ms
1,000 characters	8.073 ms	8.016 ms	5.398 ms	11.428 ms
100,000 characters	11.643 ms	11.397 ms	8.811 ms	15.241 ms
1,000,000 characters	42.600 ms	42.187 ms	35.172 ms	54.572 ms
10 properties	8.183 ms	8.115 ms	6.009 ms	11.441 ms
100 properties	7.761 ms	7.605 ms	5.782 ms	10.627 ms
1,000 properties	12.116 ms	11.566 ms	8.704 ms	16.905 ms
10,000 properties	28.612 ms	28.366 ms	24.451 ms	34.667 ms

Throughput results: The throughput results of the load test are shown in Fig. 3.8. The best average result, 93.3 RPS, was achieved by processing 10-character messages. The worst average result, 1.6 RPS, was achieved by processing 1,000,000 character messages.

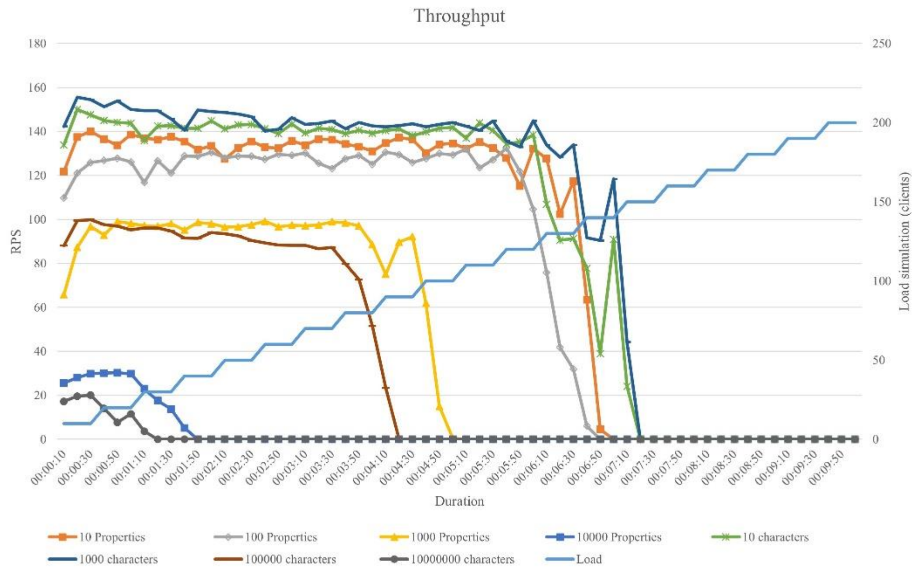


Fig. 3.8. Load test results for message processing with Kafka

Results of other metrics: Other results obtained during the experiment are presented in Table 3.6.

Table 3.6. Results of Kafka experiment measurements

Metric	Result
Request/Response size	219 B/252 B (payload 26 B)
Microservice application size	2.18 MB (empty 159 KB)
Memory usage size	40 MB (empty 9 MB)
Boot time	2.6 seconds

Architecture: To utilise Kafka as RPC, microservices have to contain two additional components: a Kafka server and a Kafka client (Fig. 3.9). The Kafka server consumes messages from topic x1 and routes them to business logic where messages are processed and moved to the Kafka client to publish them to topic y1. After pushing messages to topic y1, the Kafka client starts listening to topic y2 for a response. A message, which is consumed from topic y2, goes from the Kafka client through business logic to the Kafka server, where it is published to topic x2.

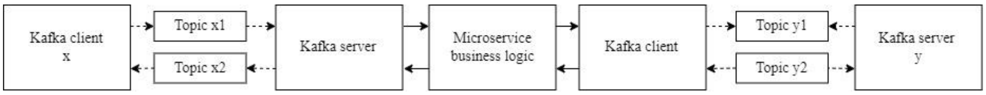


Fig. 3.9. Architecture of Kafka in microservice

Topology: The Kafka server component is not needed for those microservices that are only used as clients, and the client component is not needed for those microservices which are only used as servers. Similar to RabbitMQ, the most significant difference between HTTP Rest, gRPC and GraphQL is that there is no need for microservices to know about each other's endpoints, such as IP address or hostname. Instead of communicating directly with each other, microservices communicate through Kafka, which acts as a router. Clients are producers and produce messages to the Kafka topic while servers are consumers and consume messages from the same Kafka topic.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via Kafka technology is provided below:

- Microsoft.NETCore.App (Version 6.0.7)
- Simple.Kafka.Rpc (Version 1.8.3)

3.5.4. Evaluation Results of Google Remote Procedure Call

Latency results: Results of the latency test are shown in Table 3.7. The best results, 6.761 ms, were achieved by processing 1,000-character messages. The worst results, 35.384 ms, were achieved by processing 1,000,000 characters messages.

Table 3.7. Latency test results for message processing with gRPC

Message size	Mean	Median	Min	Max
10 characters	7.004 ms	6.787 ms	5.336 ms	9.455 ms
1,000 characters	6.716 ms	6.729 ms	5.396 ms	8.136 ms
100,000 characters	10.188 ms	10.021 ms	7.976 ms	13.537 ms
1,000,000 characters	35.384 ms	34.262 ms	25.406 ms	52.120 ms
10 properties	8.022 ms	7.929 ms	6.651 ms	9.874 ms
100 properties	8.183 ms	8.211 ms	6.692 ms	10.243 ms
1,000 properties	8.501 ms	8.487 ms	7.354 ms	10.228 ms
10,000 properties	14.855 ms	14.562 ms	12.778 ms	18.263 ms

Throughput results: The throughput results of the load test are shown in Fig. 3.10. The best average results, 170.1 RPS, were achieved by processing 1,000-character messages. The worst average result, 5.0 RPS, was achieved by processing 1,000,000-character messages.

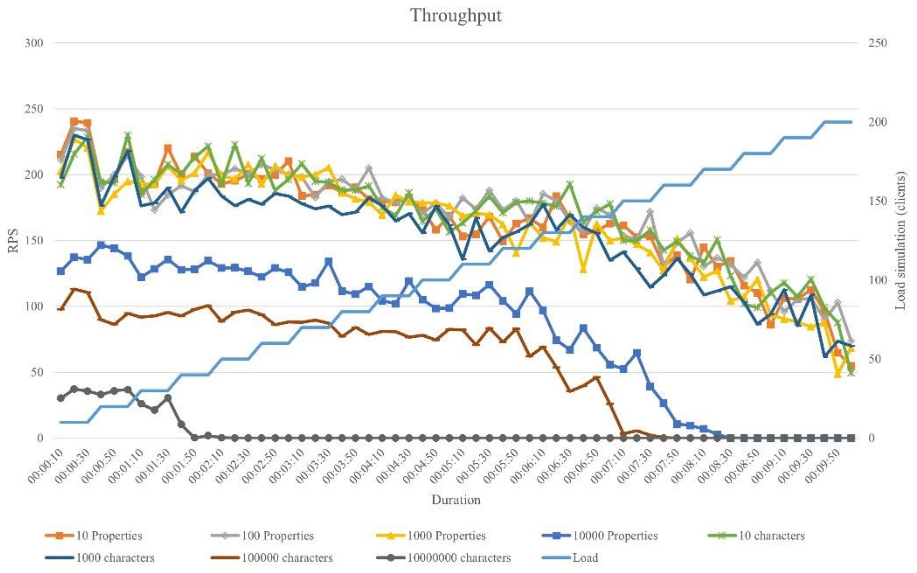


Fig. 3.10. Load test results for message processing with gRPC

Results of other metrics: Other results obtained during the experiment are presented in Table 3.8.

Table 3.8. Results of gRPC experiment measurements

Metric	Result
Request/Response size	363 B/162 B (payload 12 B)
Microservice application size	1.85 MB (empty 159 KB)
Memory usage size	70 MB (empty 9 MB)
Boot time	3.4 seconds

Architecture: To communicate via gRPC, a microservice has to have at least three additional components: gRPC server, Service, and gRPC Client (Fig. 3.11). The gRPC server component exposes the gRPC server and sends requests to the Service component, which acts as a facade for business logic. gRPC Client sends a request to gRPC server y. The components and flow are very similar to those in the Rest API case.

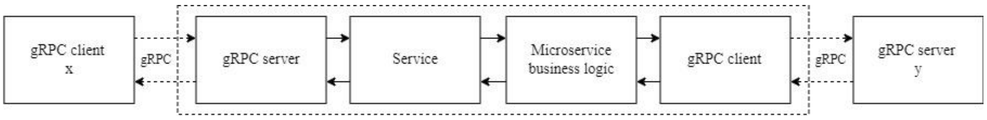


Fig. 3.11. The architecture of gRPC in microservice

Topology: In terms of topology, gRPC and Rest API have no difference. Microservices M1–M5 have to know how to reach the next microservice when a linear topology is used. Microservice M6 only exposes the gRPC server. Microservices M1, M2, and M3 have two dependencies in the tree-type topology. Microservices M4, M5, and M6 only expose the gRPC server. In the star-type topology, the M1 microservice has to know all microservice URLs.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via gRPC technology is provided below:

- Microsoft.NETCore.App (Version 6.0.7)
- protobuf-net.Grpc (Version 1.0.152)
- protobuf-net.Grpc.AspNetCore (Version 1.0.152)
- Grpc.Net.Client (Version 2.45.0)

3.5.5. Evaluation Results of GraphQL

Latency results: Results of the latency test are shown in Table 3.9. The best result, 7.711 ms, was achieved by processing 1,000-character messages. The worst result, 51.170 ms, was achieved by processing 10,000-property messages.

Table 3.9. Latency test results for message processing with GraphQL

Message size	Mean	Median	Min	Max
10 characters	7.755 ms	7.718 ms	5.945 ms	10.69 ms
1,000 characters	7.711 ms	7.376 ms	5.846 ms	12.02 ms
100,000 characters	12.349 ms	11.392 ms	9.083 ms	18.83 ms
1,000,000 characters	29.575 ms	29.137 ms	24.780 ms	38.70 ms
10 properties	10.498 ms	10.302 ms	7.652 ms	14.67 ms
100 properties	9.860 ms	9.624 ms	8.383 ms	12.63 ms
1,000 properties	13.262 ms	13.261 ms	10.921 ms	15.73 ms
10,000 properties	51.170 ms	49.828 ms	44.979 ms	65.10 ms

Throughput results: The throughput results of the load test are shown in Fig. 3.12. The best average result, 185.5 RPS, was achieved by processing 10-property messages. The worst average result, 4.8 RPS, was achieved by processing 1,000,000 characters messages.

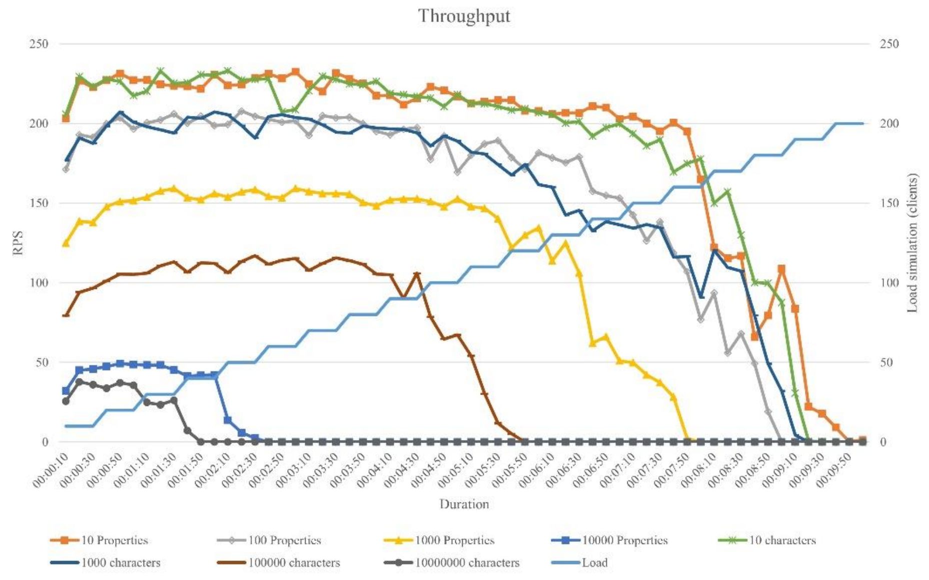


Fig. 3.12. Load test results for message processing with GraphQL

Results of other metrics: Other results obtained during the experiment are presented in Table 3.10.

Table 3.10. Results of GraphQL experiment measurements

Metric	**Result**
Request/Response size	390 B/843 B (payload 49 B)
Microservice application size	5.53 MB (empty 159 KB)
Memory usage size	65 MB (empty 9 MB)
Boot time	4.4 seconds

Architecture: GraphQL flow is quite similar to REST API. Three additional components are needed to communicate via GraphQL: GraphQL Server, GraphQL abstraction layer, and GraphQL client (Fig. 3.13). GraphQL is transport-layer agnostic, but the most common technology used for transport is HTML.

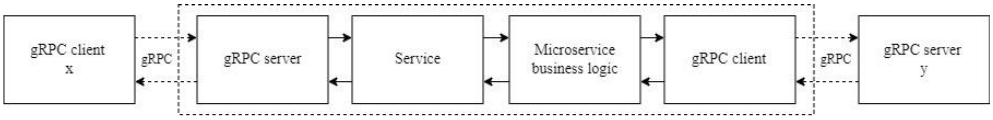


Fig. 3.13. The architecture of GraphQL in microservice

Topology: GraphQL, gRPC, and Rest API have no big difference in terms of topology. All technologies use a client/server synchronous communication model. To establish communication, a client has to know the server endpoints, such as IP address or hostname.

GraphQL is also a query language for APIs – a client can request very specific data from the server. Queries in GraphQL can be written in such a manner that would not only access separate properties but also follow references between them. Star-type topology best utilises this GraphQL feature.

Libraries: The list of libraries that were used in the experiment to establish a connection between microservices via GraphQL technology is provided below:

- Microsoft.NETCore.App (Version 6.0.7)
- RabbitMQ.Client (Version 6.3.0)
- Nito.AsyncEx (Version 5.1.2)

3.6. Comparison of Communication Technologies

This chapter compares communication technologies in different aspects based on the obtained results of the executed experiments. Chapter 3.6.1 provides details about available libraries for each technology. Chapter 3.6.2 gives an overview of the components used for each technology and highlights specific requirements for some technologies. Chapter 4.3.3 analyses the impact of the communication technology on the topology. Performance evaluation is presented in Chapter 3.6.4, using different aspects. The last sub-chapter evaluates different metrics of each technology.

3.6.1. Communication Technologies Libraries

Many different libraries can be chosen for HTTP Rest implementation mainly because it is the oldest and relatively simple technology. RabbitMQ and Kafka are also very popular technologies, so they also have quite a few libraries. GraphQL and gRPC are relatively new technologies, and not so many libraries exist in the market. Microsoft .Net framework has built-in support and provides libraries for HTTP Rest and gRPC communication technologies.

3.6.2. Communication Technologies Architecture

HTTP Rest, gRPC, and GraphQL communication technologies have very similar architecture: one component is used to expose a server, the second one is to translate from a technology-specific to business-specific message, and the last component is used to send a message.

Communication models and methods must be defined in *proto files and shared between microservices to use gRPC communication technology. Like gRPC *proto files, GraphQL has a schema. GraphQL schema contains information about server methods and data types.

RabbitMQ and Kafka are message-based technologies, and they are different from others used in the research. Communication between microservices is not point-to-point like in HTTP Rest, gRPC, and GraphQL. All communication in RabbitMQ is implemented via queues: microservices can publish to and consume from the queue. Like RabbitMQ, Kafka uses topics to implement communication. Two queues, or two topics in the Kafka case, must be created to implement RPC calls between microservices: one for a request and the second for a response.

3.6.3. Communication Technologies Topologies

HTTP Rest, gRPC, and GraphQL technologies are independent of topology. A microservice must know how to reach other microservices to establish communication, e.g., it has to know the addresses of other microservices. It is a known problem, and there are many solutions how to solve it, but all of them increase the complexity of the solution, especially if scalability is needed.

RabbitMQ and Kafka technologies do not have this challenge because they work as an intermediary communication layer, and all communication between microservices happens through it. Communication in RabbitMQ and Kafka is based on queues and topics. A microservice has to know only the name of the queue, or topic name in the Kafka case, to communicate with other microservice. A few microservices can publish and consume the same queue or topic. It is a powerful feature to support scalability.

GraphQL best utilises its features in a star-type topology where one microservice acts as a gateway and others as data sources. Powerful GraphQL query language allows the creation of a specific request in such a way that it can fetch data from multiple data sources in one API call. This feature can potentially reduce the number of calls between microservices needed to implement the functionality.

3.6.4. Communication Technologies Performance

Performance tests were executed to compare latency and throughput in the case of RPC calls between five microservices. No performance optimisations were applied to any technology during this experiment. Latency results based on message size in characters are shown in Fig. 3.14. Latency results based on several properties are shown in Fig. 3.15.

The lowest latency results for strings up to 1,000,000 characters were obtained by RabbitMQ technology. RabbitMQ RPC calls were two times faster than other technologies. It showed the best results for processing the smallest messages (ten and 1,000 characters); the results were two times better than processing 100,000-character messages. HTTP Rest, Kafka, gRPC and GraphQL showed similar latency results; however, results obtained by gRPC were slightly better.

On the other hand, the RabbitMQ had the highest latency results while processing messages which consisted of 10,000,000 characters. It was from three to four times slower than others. The best latency results for 10,000,000-character messages were obtained by GraphQL and HTTP Rest technologies. Kafka was 40% and gRPC was 16% and slower than GraphQL and HTTP Rest technologies.

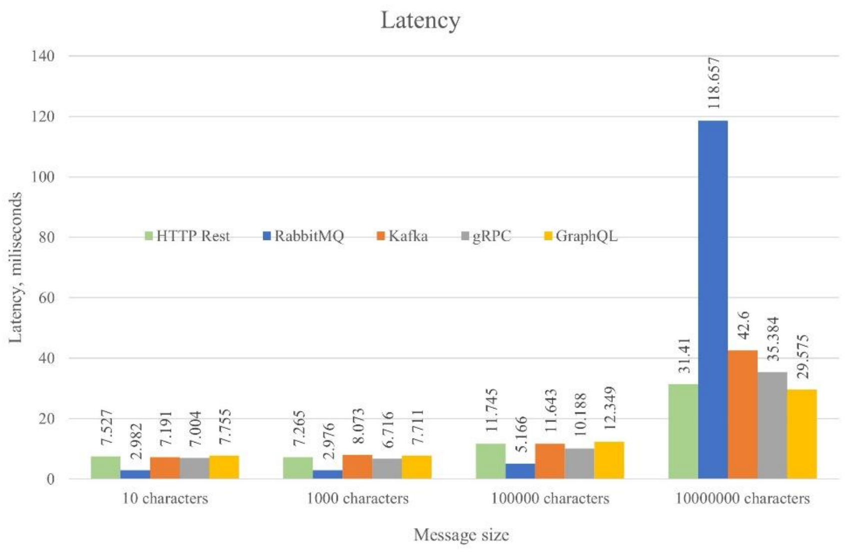


Fig. 3.14. Latency test results based on string size

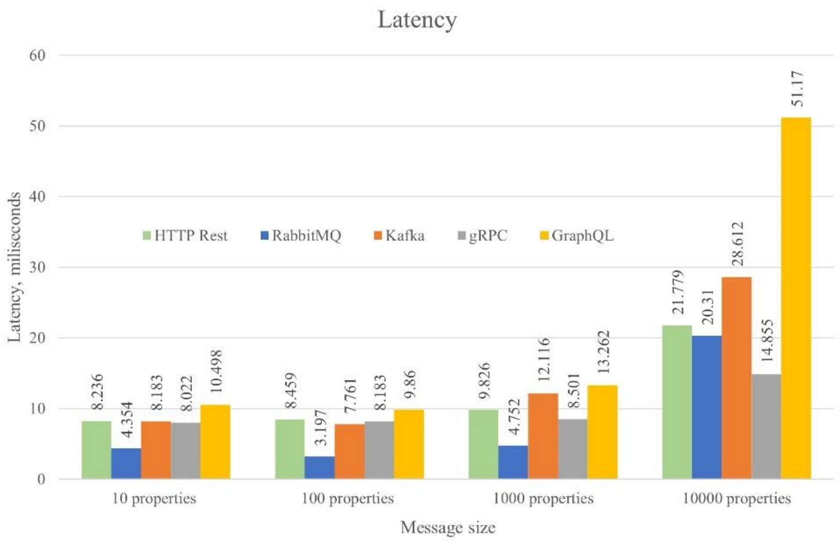


Fig. 3.15. Latency test results based on string size

The lowest latency results for messages containing up to 1,000 properties were also obtained by RabbitMQ technology. RabbitMQ RPC calls were two to three times faster than other technologies. It showed the best results for processing messages containing 100 properties; the results were 37% better than processing messages containing 1,000 properties and 47% better than processing messages containing ten properties. HTTP Rest, Kafka and gRPC showed similar results for messages containing ten and 100 properties. The best results for communicating via messages containing 10,000 properties were obtained by gRPC technology. The binary serialisation used by gRPC technology is faster than JSON serialisation, which has been used by other technologies during the experiment; hence, the more properties the message contains, the greater advantage gRPC has. The GraphQL showed the worst latency results for messages containing at least ten properties. The more properties the message contained, the greater the difference was compared to other technologies. It was from two to four times slower than others while communicating via messages containing 10,000 properties. Analysis of the results shows that RabbitMQ achieved the best RPC call latency results in six out of eight cases. However, the RabbitMQ was the slowest technology, processing 10,000,000 characters of messages. It can be summarised that the RabbitMQ has the lowest latency if the message size is not bigger than 0.1MB and the data model contains up to 1,000 properties.

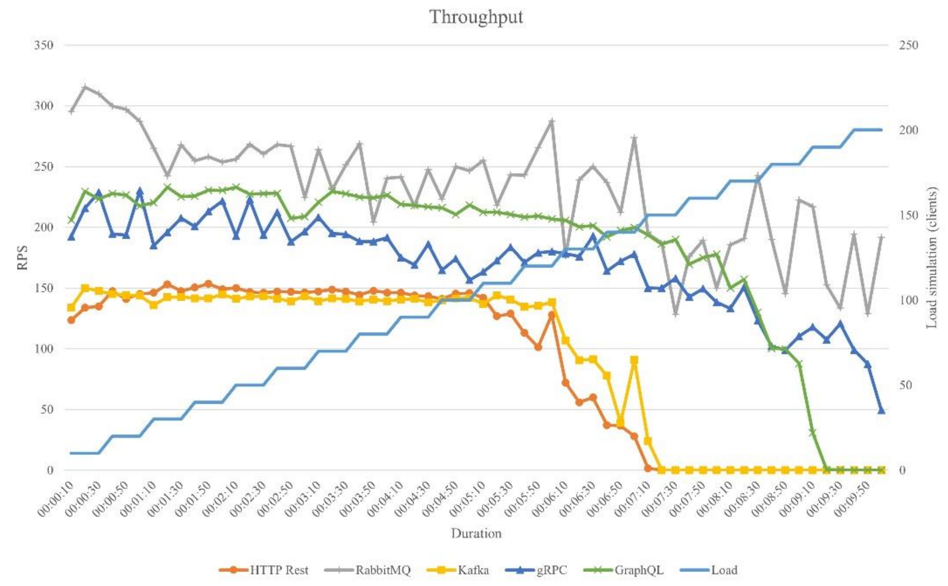


Fig. 3.16. Throughput test result for 10-character size messages

Throughput results for 10-character-size messages are shown in Fig. 3.16. The best throughput results were obtained by RabbitMQ technology, with an average of 231.6 RPS. The maximum result, 315.1 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by HTTP Rest technology with an average of 89.8 RPS and a limit of 140 clients.

Throughput results for 1,000 character-size messages are shown in Fig. 3.17. The best throughput results were obtained by RabbitMQ technology, with an average of 219.5 RPS. The maximum result, 300.1 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by HTTP Rest technology with an average of 89.9 RPS and a limit of 140 clients.

Throughput results for 100,000 character-size messages are shown in Fig. 3.18. The best throughput results were obtained by RabbitMQ technology, with an average of 93.3 RPS. The maximum result, 179.3 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by Kafka technology, with an average of 36.2 RPS and a limit of 80 clients.

Throughput results for 10,000,000 characters size message are shown in Fig. 3.19. The best throughput results were obtained by gRPC technology, with an average of 5.0 RPS and a limit of 40 clients. The maximum result, 37.1 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by RabbitMQ technology with an average of 0.01 RPS.

Throughput results for ten properties size messages are shown in Fig. 3.20. The best throughput results were obtained by RabbitMQ technology, with an average of 200.4 RPS. The maximum result, 291.5 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by Kafka technology, with an average of 87.0 RPS and a limit of 140 clients.

Throughput results for 100 properties size messages are shown in Fig. 3.21. The best throughput results were obtained by RabbitMQ technology, with an average of 203.5 RPS. The maximum result, 295.5 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by Kafka technology, with an average of 72.2 RPS and a limit of 130 clients.

Throughput results for 1,000 properties size messages are shown in Fig. 3.22. The best throughput results were obtained by gRPC technology, with an average of 161.9 RPS. The maximum result, 227.0 RPS, was reached while requesting ten clients. The worst RPC throughput test results were obtained by Kafka technology, with an average of 43.7 RPS and a limit of 100 clients.

Throughput results for 10,000 properties size messages are shown in Fig. 3.23. The best throughput results were obtained by gRPC technology, with an average of 83.3 RPS. The maximum result, 146.6 RPS, was reached while requesting 20 clients. The worst RPC throughput test results were obtained by Kafka technology, with an average of 3.9 RPS and a limit of 30 clients.

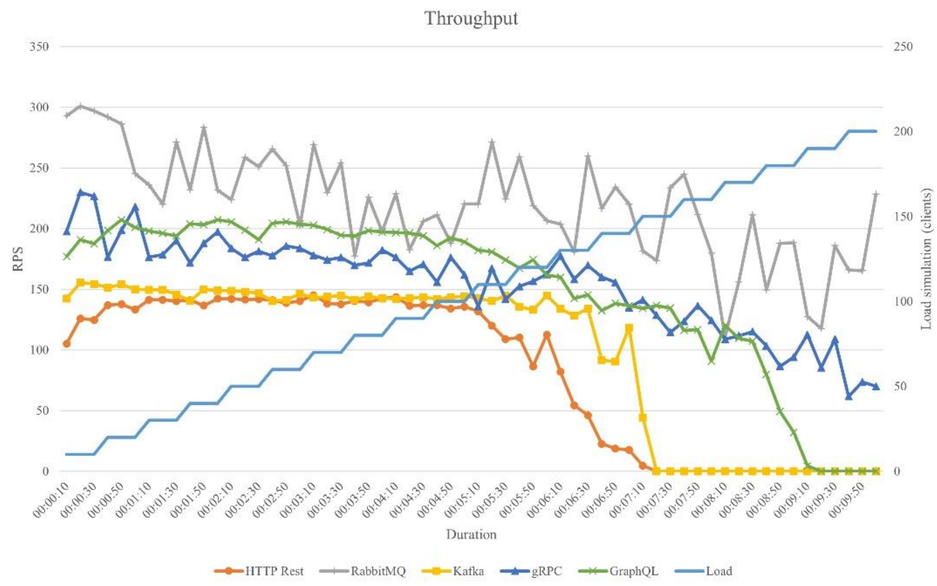


Fig. 3.17. Throughput test results for 1,000-character-size messages

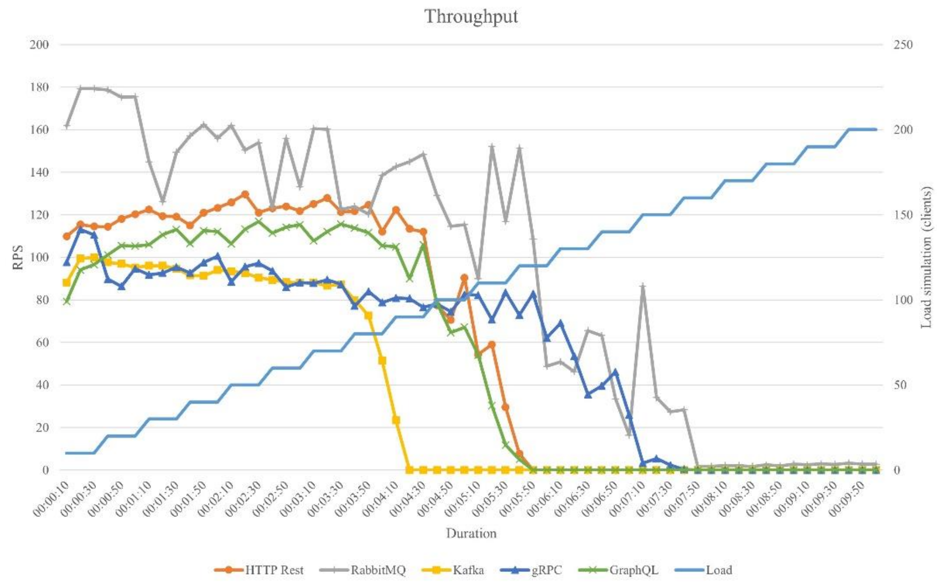


Fig. 3.18. Throughput test results for 100,000-character-size messages

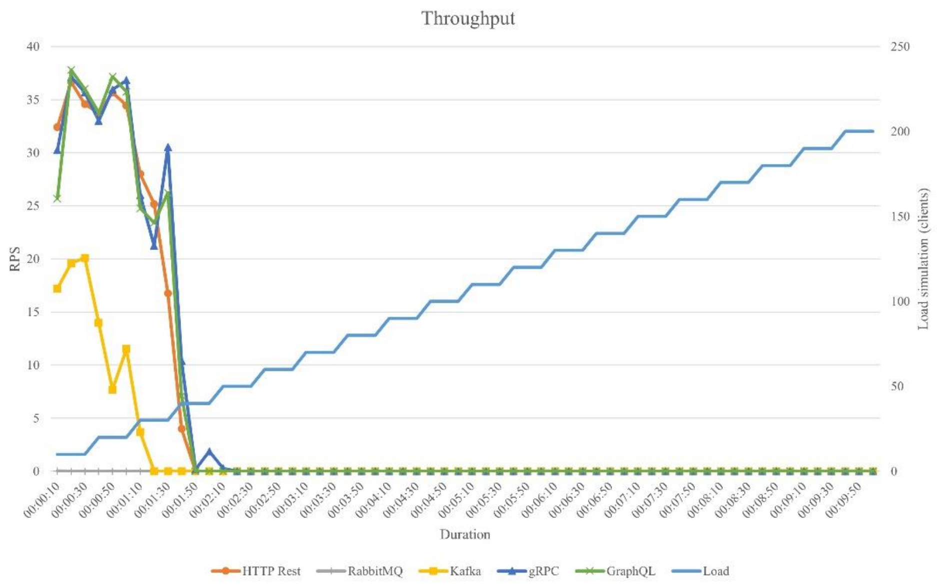


Fig. 3.19. Throughput test results for 10,000,000-character-size messages

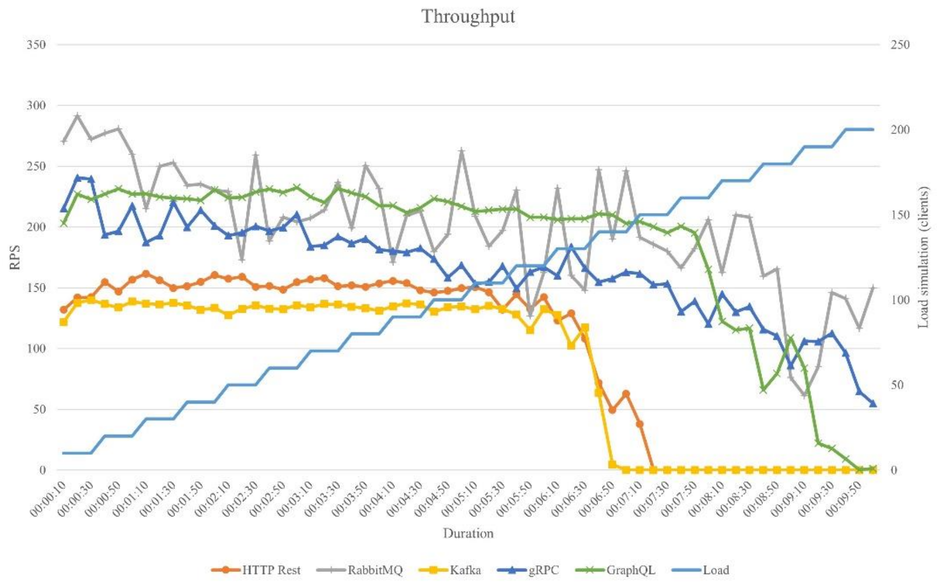


Fig. 3.20. Throughput test results for 10-property-size messages

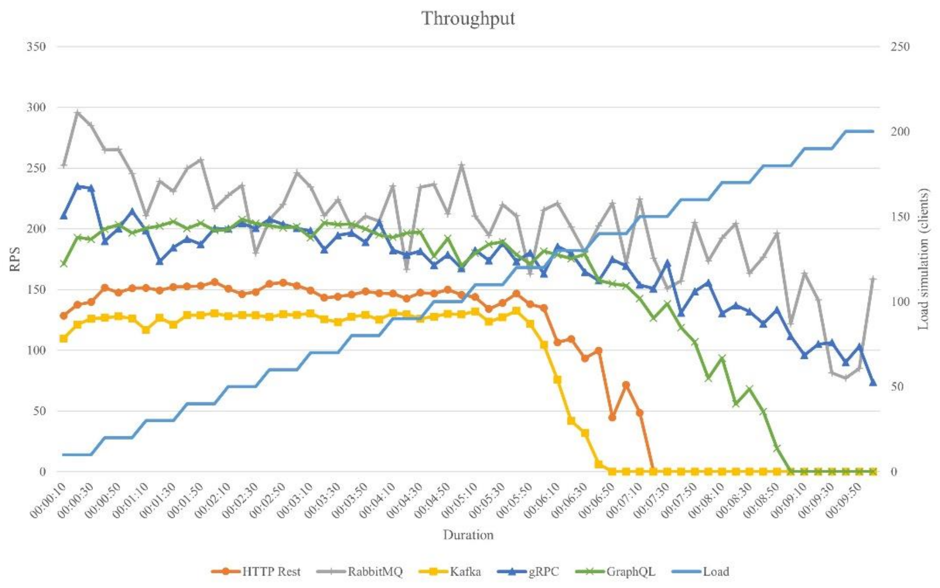


Fig. 3.21. Throughput test results for 100-property-size messages

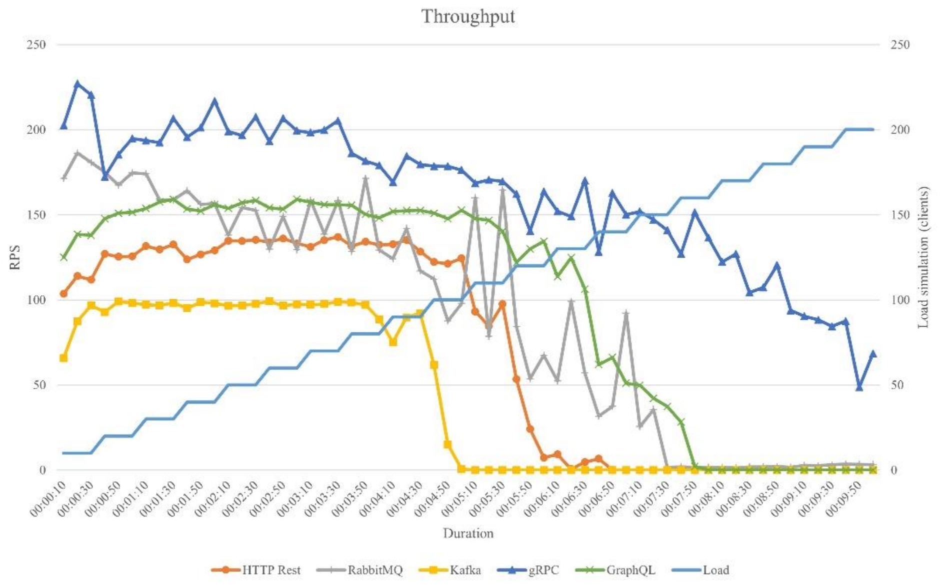


Fig. 3.22. Throughput test results for 1,000-property-size messages

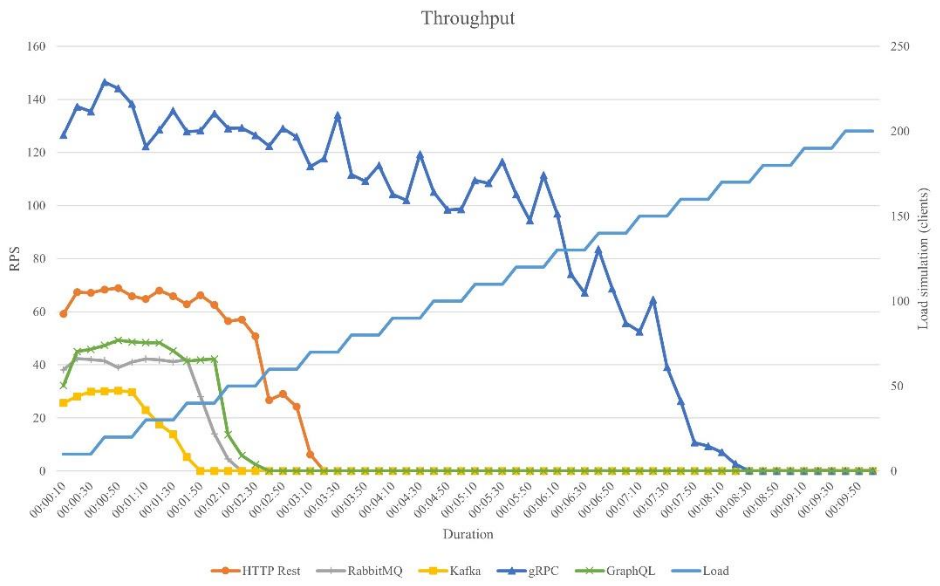


Fig. 3.23. Throughput test results for 10,000-property-size messages

It can be summarised that the best RPC call throughput results for smaller messages, up to 0.1MB and up to 100 properties, were achieved by RabbitMQ technology. The best RPC call throughput results for bigger messages were achieved by gRPC communication technology. The worst throughput results in five of eight cases were achieved by Kafka.

However, latency distribution results (Figs. 3.24–3.28) show that both Kafka and RabbitMQ can process more messages (with latency higher than one second) and work more stable when dealing with more than 50 clients load, compared to HTTP Rest, gRPC and GraphQL technologies.

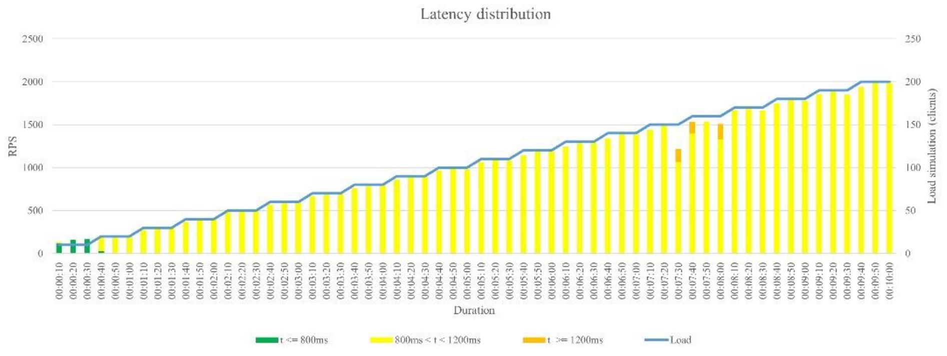


Fig. 3.24. Kafka latency distribution for 1,000,000-character-size messages

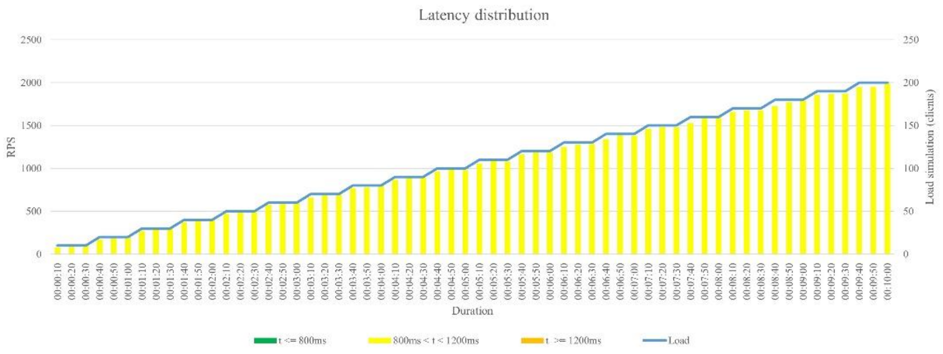


Fig. 3.25. RabbitMQ latency distribution for 1,000,000-character-size messages

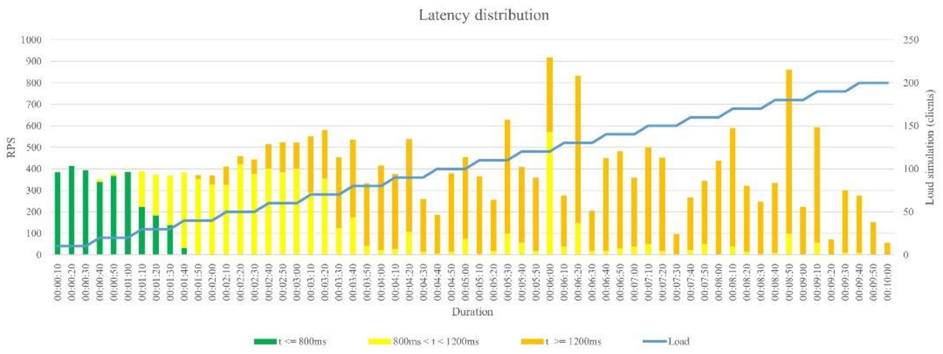


Fig. 3.26. HTTP Rest latency distribution for 1,000,000-character-size messages

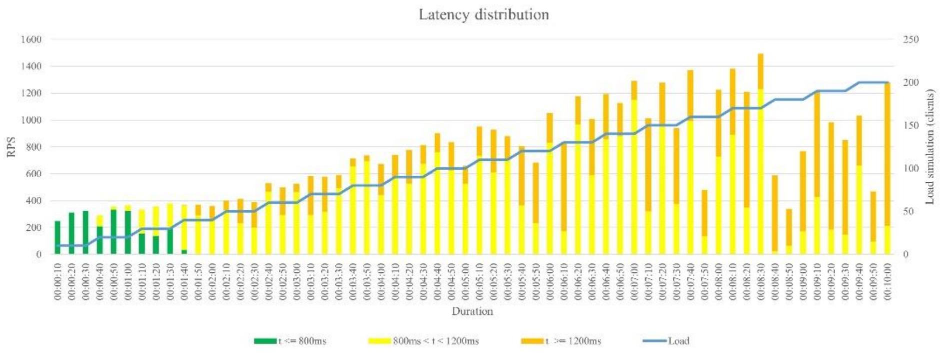


Fig. 3.27. gRPC latency distribution for 1,000,000-character-size messages

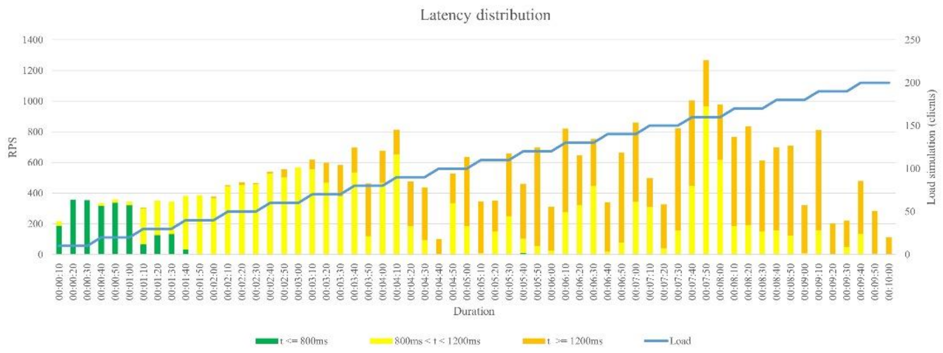


Fig. 3.28. GraphQL latency distribution for 1,000,000-character-size messages

The latency distribution results reveal that Kafka and RabbitMQ outperform HTTP Rest, gRPC, and GraphQL technologies in terms of stability and processing capacity for messages with higher latency, especially under heavy client load.

3.6.5. Communication Technologies Metrics

The smallest size of request/response was obtained by HTTP Rest technology, with a total size of 357 B. The GraphQL request/response was approx. 2–3 times bigger than others (Fig. 3.29). If the message size is an important criterion when choosing communication technology, then HTTP Rest is a recommended technology. On the other hand, GraphQL supports remote querying, so potentially, one GraphQL request/response could transfer as much information as a few requests/responses using other technologies.

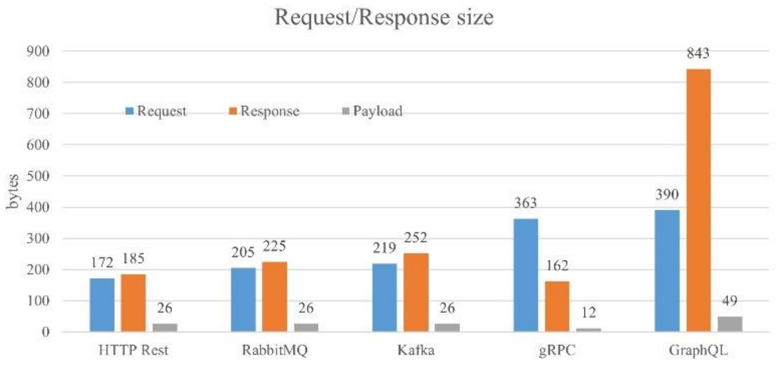


Fig. 3.29. Request/Response size measured during the experiment

A comparison of application size is presented in Fig. 3.30. It can be seen that the biggest application size of 5530 KB was obtained when GraphQL libraries were used for microservices. The smallest application size of 1850 KB was when GraphQL libraries were included. Application size is independent of communication technology. It depends on how it was implemented in the library. If the library size is too big, then the microservice developer can implement it by him selves.

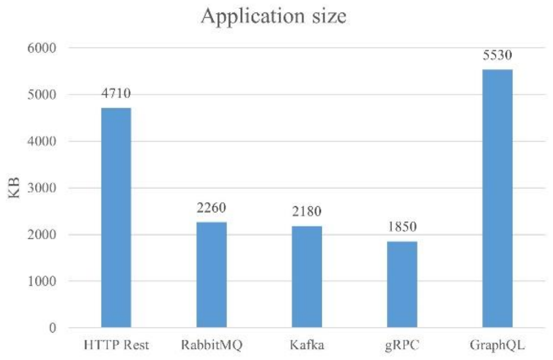


Fig. 3.30. Application size measured during the experiment

The smallest amount of memory, 23 MB, was allocated using RabbitMQ libraries, while gRPC used 70 MB of memory, which is almost three times more than RabbitMQ (Fig. 3.31). It can be noted that if an application is running in an environment where memory is limited, then the best solution for implementing communication is between RabbitMQ and Kafka. Also, it must be pointed out that RabbitMQ and Kafka do require additional applications compared to other communication technologies.

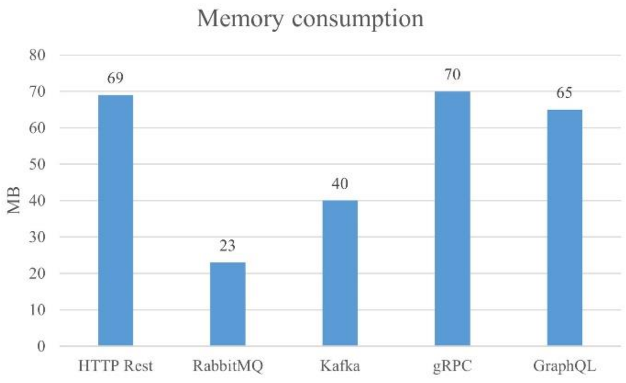


Fig. 3.31. Memory consumption measured during the experiment

A comparison of microservice boot time is shown in Fig. 3.32. The longest boot time was spotted using GraphQL technology, and it took 4.4 seconds, while the shortest boot time of 2.6 seconds was obtained using Kafka technology. Boot time, as well as the microservice size, mostly depend on implementation, but not on communication technology itself and can be potentially improved by tuning implementation details.

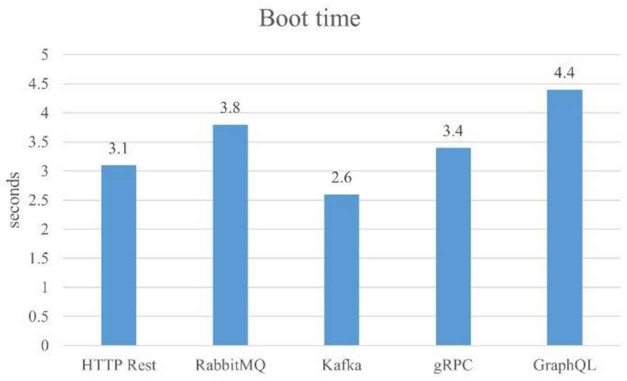


Fig. 3.32. Boot time measured during the experiment

HTTP Rest technology is optimal for smaller request/response sizes, while GraphQL, despite larger sizes, offers robust remote querying capabilities. Application size, influenced more by library implementation than communication technology, can vary significantly with GraphQL libraries. Memory allocation is lowest with RabbitMQ libraries, making them suitable for memory-limited environments, though RabbitMQ and Kafka do require additional applications. Microservice boot time, primarily dependent on implementation rather than communication technology, is longest with GraphQL and shortest with Kafka.

3.7. Conclusions of the Third Chapter

One of the most significant challenges during the monolith application transition into microservice architecture is data communication management. How should migration from process method or function calls to inter-process communication be done? The high complexity, variety of architectural aspects, technological stack, and business objects make every application different and create challenges during monolith application decomposition to microservices. The introduced criteria allow for the evaluation of various aspects of communication technologies that are important while designing microservices. The key findings discovered in this research are provided below:

1. If latency and throughput are the main criteria during the transition from a monolith architecture to a microservice architecture, then RabbitMQ and gRPC are the most suitable technologies. RabbitMQ showed the best results in RPC latency and throughput tests for small messages (up to 0.1MB and data model up to 100 properties), while gRPC showed the best results in RPC latency and throughput tests for big messages. The worst result was obtained by HTTP Rest and Kafka technologies.
2. Kafka and RabbitMQ showed the best throughput results in the most loaded conditions: requested by more than 100 clients at the same time and processing 1,000,000 characters of messages. However, the latency of RPC was high, more than one second.
3. If horizontal scalability is an important aspect, Kafka and RabbitMQ are the best candidates as they have built-in cluster functionality. It must be noted that other technologies can be scaled horizontally as well, but it requires additional tools and effort.
4. HTTP Rest has the smallest request and response message size. If the message size is an important criterion when choosing communication technology, then HTTP Rest is a recommended technology. On the other hand, gRPC has the smallest payload as it uses binary serialisation. Theoretically, at some point of complexity, for complex data models with many properties, gRPC request and response message size should become smaller than HTTP Rest. Deeper research is needed to determine the exact complexity threshold.
5. The gRPC library uses the least amount of storage. If microservices are running in an environment with limited storage, then gRPC must be used. The maximum amount of storage is allocated for GraphQL libraries. It must be pointed out that storage size weakly depends on technology. It mostly depends on how it was implemented in the particular library. If the library size is too big, then microservice developers can implement it by themselves, but there is no guarantee that the new library will be smaller.
6. RabbitMQ and Kafka consume the smallest amount of memory. Therefore, if memory size is one of the essential criteria, then RabbitMQ and Kafka must be used for implementation. On the other hand, HTTP Rest consumes the largest amount of memory. Memory size and storage usage depend on library implementation, so a similar recommendation can be provided to the previous item on the list.
7. Microservice implemented using Kafka library boots up in the fastest way while using GraphQL library boots up in the slowest way. If the boot time or restart time of the microservice is essential, then Kafka must be used for microservice communication.

8. HTTP Rest and RabbitMQ are prevalent communication technologies, and many different libraries exist in the market to choose from, while GraphQL and gRPC are relatively new and rapidly growing communication technologies with fewer libraries to choose from.
9. Synchronous communication style communication technologies gRPC, HTTP Rest, and GraphQL do not require any additional components to communicate, while asynchronous communication technologies RabbitMQ and Kafka require service as an interim communication layer. Hence, additional components increase solution complexity and maintenance costs. On the other hand, if a solution contains many microservices and scalability is a challenge, RabbitMQ and Kafka as an interim layer can provide centralised communication routing functionality.

Known limitations and threats to the validity of the conducted research are provided below:

1. The experiment was conducted using the programming language C Sharp. Measured results can be different using other programming languages and libraries.
2. The experiment was conducted using a computer with Windows OS. Measured results can be different when using different environments such as Linux, Docker, OpenShift, public cloud, etc., due to their specifics and the implementation details of the libraries.

4

The approach of Monolith Database Migration into Multi-Model Polyglot Persistence

Migration from a monolithic architecture to a microservice architecture is a complex challenge that consists of issues such as microservice identification, code decomposition, a combination of 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 a microservice architecture, data storage is decentralised, and 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's need.

This chapter evaluates the proposed approach of monolith database migration into multi-model polyglot persistence based on microservice architecture. The novelty and relevance of the proposed approach are double; e.g., it provides a general approach to conducting database migration from a 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 could be used to conduct data storage migration from a monolith to a microservice architecture and improve the quality of the consistency, understandability, availability, and portability attributes. The purpose of the proposed approach is shown in Fig. 4.1. A detailed explanation of the proposed migration approach is provided in Chapter 2.5.

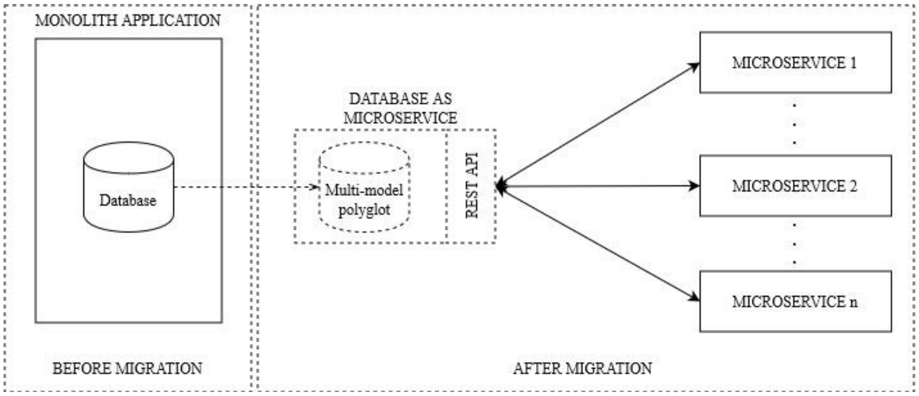


Fig. 4.1. Purpose of the proposed database migration approach

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 utilise the benefits of microservices, such as agility and scalability. The encapsulation of a database into a microservice reduces the complexity and increases the performance. 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 gradually from the monolith architecture to the microservice architecture without considering the database that has already been 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.

The proposed approach and results presented in this chapter were published in the author’s publication (Kazanavicius, Mazeika, Kalibatiene et al., 2022).

4.1. Evaluation Criteria of the Approach of Monolith Database Migration into Multi-Model Polyglot Persistence

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.

4.2. Multi-Model Polyglot Database Software

ArangoDB is an open-source multi-model polyglot persistence system that implements a data model integrating document, graph, and key–value models with one database core. It supports transactions, partitioning, and replication (ArrangoDB, 2023). ArangoDB has its query language AQL, which allows joins, operations on graphs, iterations, filters, projections, ordering, grouping, aggregate functions, union, and intersection. The ArangoDB supports all the ACID properties.

Table 4.1. Comparison of multi-model polyglot databases

Database	**Document**	**Graph**	**ACID**	**SQL**	**AQL**	**C#**	**On premise**
ArrangoDB	Yes	Yes	Yes	No	Yes	Yes	Yes
Azure Cosmos DB	Yes	Yes	Yes	Yes	No	Yes	No
CrateDB	Yes	No	Yes	Yes	No	Yes	Yes
EnterpriseDB	Yes	No	Yes	Yes	No	Yes	Yes
MarkLogic	Yes	Yes	Yes	Yes	No	Yes	Yes
OrientDB	Yes	Yes	Yes	Yes	No	Yes	Yes
SAP HANA	Yes	Yes	Yes	Yes	No	Yes	No
Virtuoso	Yes	Yes	Yes	Yes	No	No	Yes

The most important criteria used to choose multi-model polyglot database are Table 4.1. Even though ArrangoDB does not support SQL it has AQL support which better utilises multi-model polyglot persistence features and advantages as it supports various data formats or patterns.

4.3. Tools Used to Evaluate the Approach of Monolith Database Migration into Multi-Model Polyglot Persistence

The ArangoDB community edition version 3.7.11 database was used as the database engine. The microservice that exposes multi-model polyglot persistence was written using C#.NET5 framework. All coding and testing were done using Microsoft Visual Studio IDE and Arango Management Interface. All libraries used in the research were downloaded from the NuGet gallery. The experiment was performed on a computer with the following specifications: CPU – Core i7 9850H, memory – 32 GB RAM, storage – 512 GB SSD, and OS – Windows 10 Enterprise. All applications were run on a computer, and no external devices or networks were used.

4.4. Evaluation results of the Approach of Monolith Database Migration into Multi-Model Polyglot Persistence

This chapter provides results obtained during the evaluation of the method of 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 sub-chapters: 4.4.1. Analysis of an Existing Monolith Application with a Mainframe Database, 4.4.2. Data Model Development, 4.4.3. Microservice Development, 4.4.4. Data Transformation, 4.4.5. Data Validation, and 4.4.6. Release and Deployment.

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

The primary function of the SSI application is to store and provide standard settlement instructions to other information systems across the organisation. Standard settlement instructions are used to execute payments between banks and organisations. A simplified model of the SSI application is shown in Fig. 4.2.

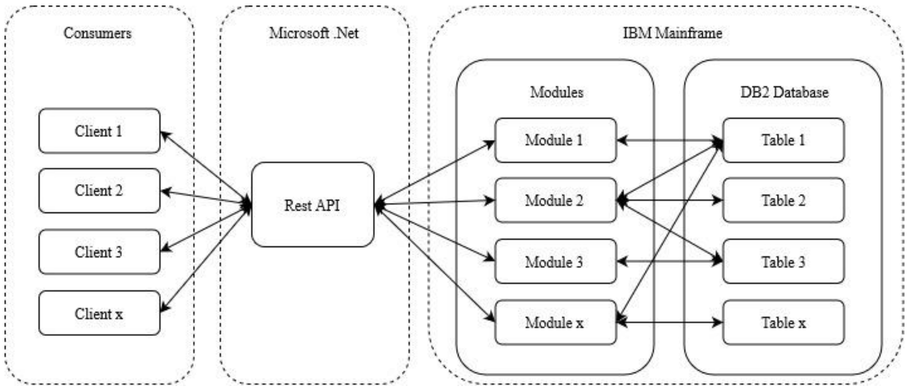


Fig. 4.2. Simplified model of the SSI application

This SSI application is implemented with IBM mainframe and Microsoft .Net framework technologies. The data is persisted in 35 tables in the DB2 database, and it can be accessed and edited through IBM mainframe modules. SSI data is exposed to other information systems across the organisation through Rest API, which is implemented with the Microsoft .Net framework. The most important functional requirements gathered during the evaluation are presented in Table 4.2.

Table 4.2. Functional requirements of 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

Using a top-down approach, functional requirements were collected in two steps. Firstly, essential features were identified through discussions with domain experts. Secondly, a thorough review of the legacy code was conducted, which provided insights into existing practices and highlighted areas for improvement or reuse. This approach ensured a comprehensive understanding of the system's needs.

4.4.2. Data Model Development

This step aims 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.

4.4.2.1. Conceptual Design

The conceptual design step aims 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 in the database. The simplified conceptual database schema of the SSI application is shown in Fig. 4.3.

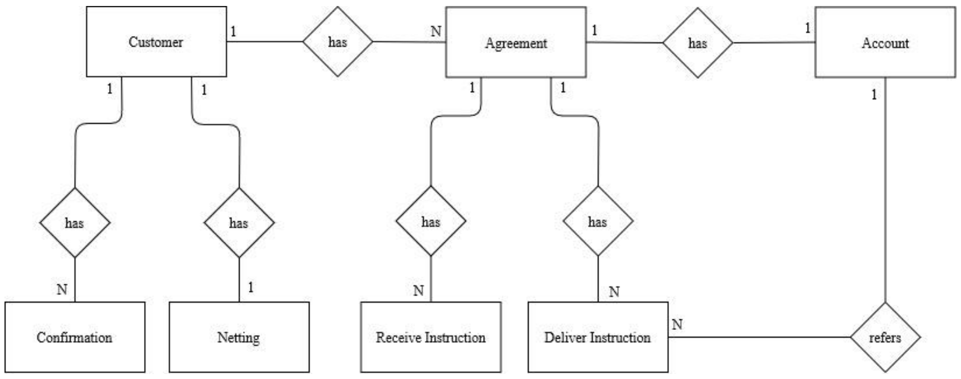


Fig. 4.3. Simplified conceptual schema of the SSI application

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 has 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.

4.4.2.2. Segmentation Design

The segmentation step identifies independent functional units and defines the borders between them. Segmentation units have to be identified 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 the 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 Fig. 4.4.

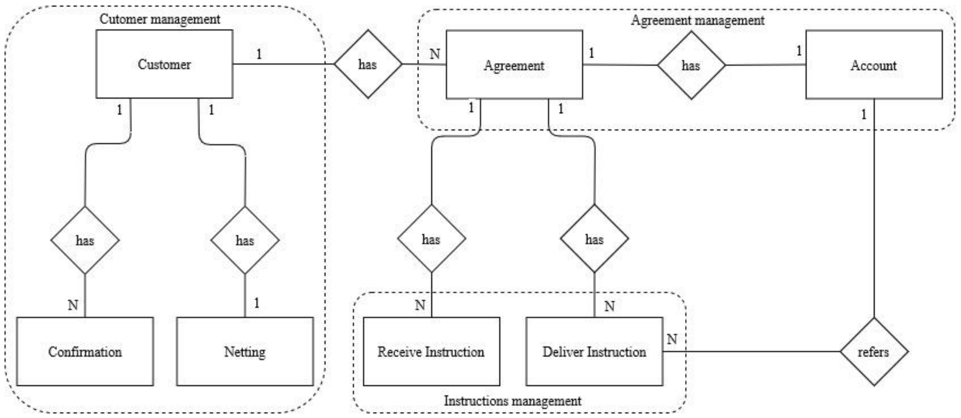


Fig. 4.4. Segmentation units are identified in the simplified conceptual database schema

During the segmentation design sub-step, the SSI application was divided into three independent functional units: customer management, agreement management, and instruction management.

4.4.2.3. Consistency Design

The consistency step ensures the dataset's consistency across all subsystems and allows for data fragmentation. As polyglot supports NoSQL data models, the eventual consistency provided by BASE properties has to be considered 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.

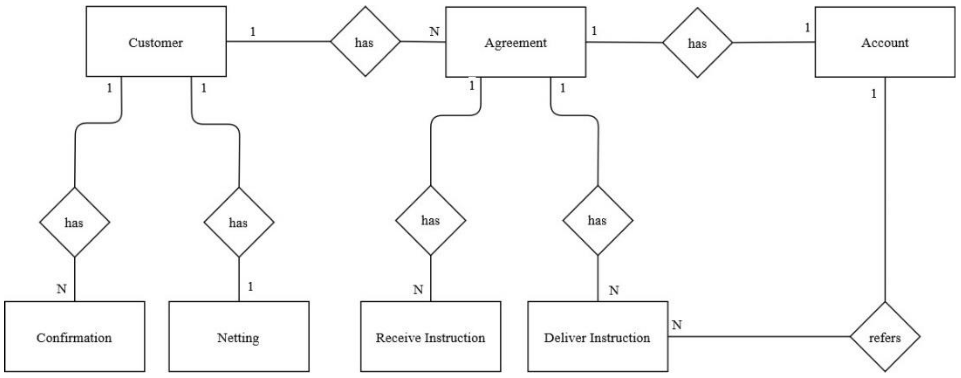


Fig. 4.5. Consistency unit identified in the simplified conceptual database schema

The consistency unit must guarantee that all reads of the entity will eventually return the last updated value, provided no new updates are made to an entity. An example of one consistency unit in the SSI application is shown in Fig. 4.5. Customer, agreement, and receive instruction comprise a consistency unit, and in the case that a query returns the response with different versions of items, an inconsistency arises that may cause a system failure.

4.4.2.4. Target 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 models. The identified target data model is shown in Fig. 4.6.

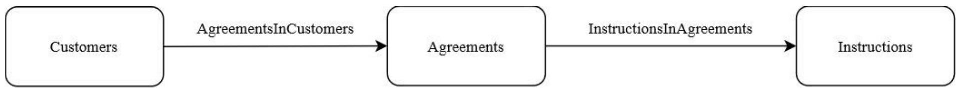


Fig. 4.6. Target data model

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 stored in separate collections.

4.4.2.5. Physical Design

The physical design step aims to implement all peculiarities of the planned-to-use database 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 as key–value pair. Documents are grouped into collections. ArangoDB supports two types of collections: document collections 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 Fig. 4.7. and its representation as a graph, where customer has two agreements, is shown in Fig. 4.8.

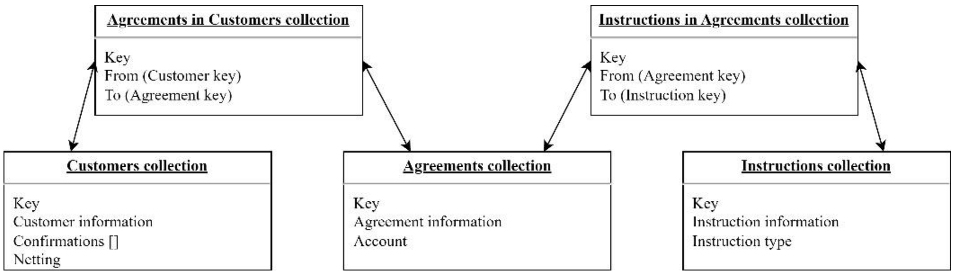


Fig. 4.7. Target data model

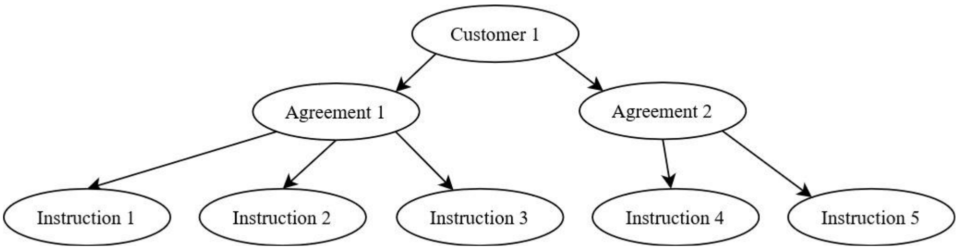


Fig. 4.8. Graphical representation of the physical data model

The physical model consists of a document collection: (1) Customers – to store the customer data, (2) Agreements – to store the agreement data, and (3) Instructions – to store the instruction data. To create relations between the documents, two edge collections were introduced: (1) AgreementsInCustomers – to store the relations between a customer and its agreements, and (2) InstructionsInAgreement – to store the relations between an agreement and its instructions.

4.4.3. Microservice Development

The simplified model of the built microservice with multi-model polyglot persistence is shown in Fig. 4.9.

The pattern database as a service was chosen to be used 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 of scaling up on demand automatically. The ArangoDB was used as a multi-polyglot database, and its cluster was established with two nodes, one per availability zone. Security and accessibility were ensured by firewall rules and separate access rights for specific operations.

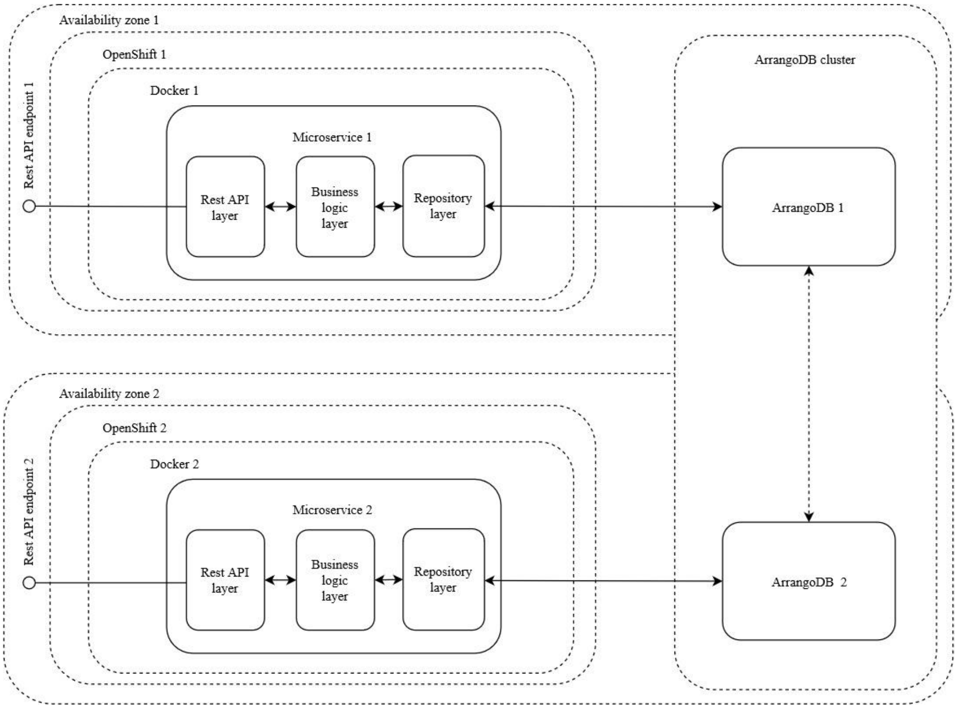


Fig. 4.9. Simplified model of a new SSI application with multi-model polyglot persistence

In the new SSI application, a database and a business logic worked as one unit – the SSI microservice. The data are exposed to other information systems across an organisation 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 of 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.

4.4.4. Data Transformation

A data transformation application was written with the C# programming language (Fig. 4.10) for migrating data from a monolith database to a multi-model polyglot persistence database. 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.

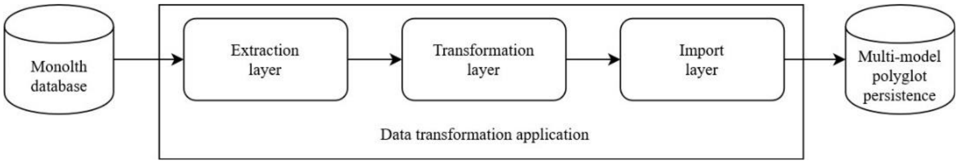


Fig. 4.10. Data transformation from the monolith database to multi-model polyglot persistence

The data was extracted from 35 tables in the IBM DB2 database, transformed, and imported into three document collections and two edge collections. This complex process was meticulously designed to ensure data integrity and consistency across both databases.To make sure that the data is consistent in both databases, the actual data transformation was conducted during the release and deployment steps. The mainframe application was temporarily stopped to transform the data. This pause allowed the team to carry out the data transformation effectively, ensuring there were no active changes happening in the database while the process took place. After successfully transforming the data, necessary amendments were made to the system configuration to start using the microservice instead of the existing monolith database.

4.4.5. 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 organisation. The forum consisted of three SSI domain experts, three mainframe software engineers, and four C# software engineers. A test engine was written with the C# programming language to execute automatic data validation (Fig. 4.11) and contained three modules.

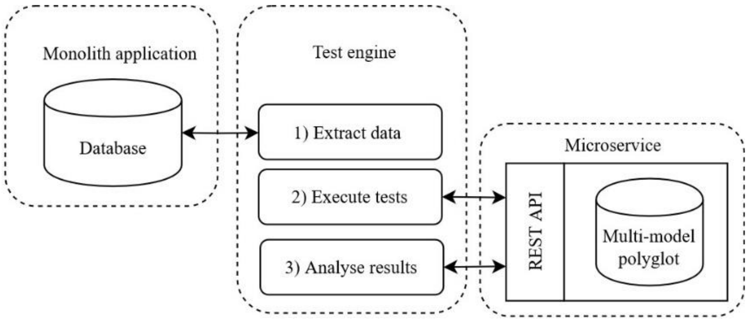


Fig. 4.11. Automatic data validation process

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 requests customer data, one by one, from Microservice REST API, and the analysis module validates that all customers exist in multi-model polyglot persistence.

4.4.6. 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-premises cloud as a Docker container to OpenShift. Four instances of microservice were distributed between two microsegments, two instances in each microsegment. Each microsegment was in different data centres. Microservice deployment into the cloud schema ensures a high resilience and availability level (Fig. 4.12). Kubernetes ensure resilience for containers in each microsegment and the distribution between two microsegments ensures high availability. A load balancer provides one point for the clients to the REST API. The continuous integration (CI) and continuous deployment (CD) pipelines were created in Azure DevOps.

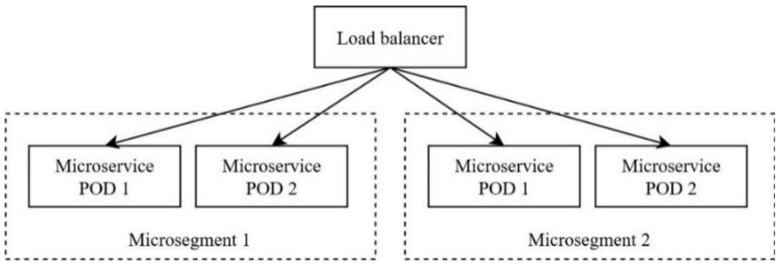


Fig. 4.12. Automatic data validation process

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 hyper-care period started. Once the hyper-care was over, the legacy monolith mainframe database was decommissioned.

4.5. 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 chapter 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 4.3.

Table 4.3. 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 organisation. The forum consisted of two SSI domain experts, four mainframes software engineers, and four C# software engineers. To ensure the reliability of experts, a two-part verification has been conducted. At first, it was ensured that the experts had relevant knowledge and at least five years of experience in the domain. Secondly, experts had to pass interviews that allowed them to ensure the sufficiency of their knowledge in relevant domains. Proof-of-the 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, ten questions for each quality attribute. Each question had to be applied to both applications. Questionnaires were constructed in a way that made 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), and *strongly agree* (5 points). Fleiss' kappa κ inter-rater agreement was used to assess the agreement among the experts (Fleiss et al., 2003). The coefficient value was 0.77, which indicates a relatively high level of agreement between the experts. If the test statistic κ was 1, then all of the survey respondents were unanimous, and each respondent was assigned the same rate to the list of concerns. If κ 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 κ indicate a greater or lesser degree of unanimity among the various responses.

In Table 4.4, the conclusive outcomes derived from the comprehensive evaluation and comparison process are meticulously displayed. The final value of each quality attribute is a calculated average, precisely rounded to the nearest whole number, based on the collective opinions of the experts involved.

Table 4.4. 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

End of Table 4.4

Quality Attribute	Monolith	Microservice
Precision	5	5
Traceability	5	5
Understandability	3	5
Availability	2	4
Portability	1	5
Recoverability	4	4

The meticulous portrayal of the evaluative landscape extends further as Fig. 4.13 is scrutinised, providing a comprehensive visual analysis of the ISO/IEC 25012:2008 standard quality attributes tailored to the intricacies of the monolith mainframe application.

Simultaneously, Fig. 4.14 delves into the corresponding assessment for the microservice application, amplifying the scientific rigour applied to the evaluation process. The graph meticulously elucidates the distribution patterns of expert responses, underlining the nuances inherent in their qualitative judgments. Furthermore, the inclusion of standard deviation metrics serves as a pivotal component, enhancing the robustness of the analysis by providing insights into the degree of variability among expert opinions.

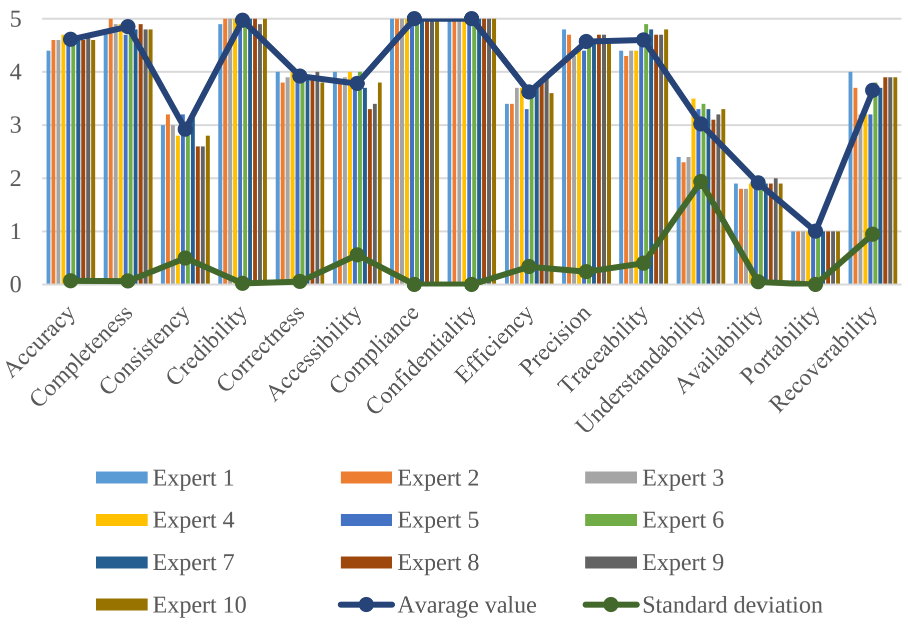


Fig. 4.13. Evaluation results of the monolith

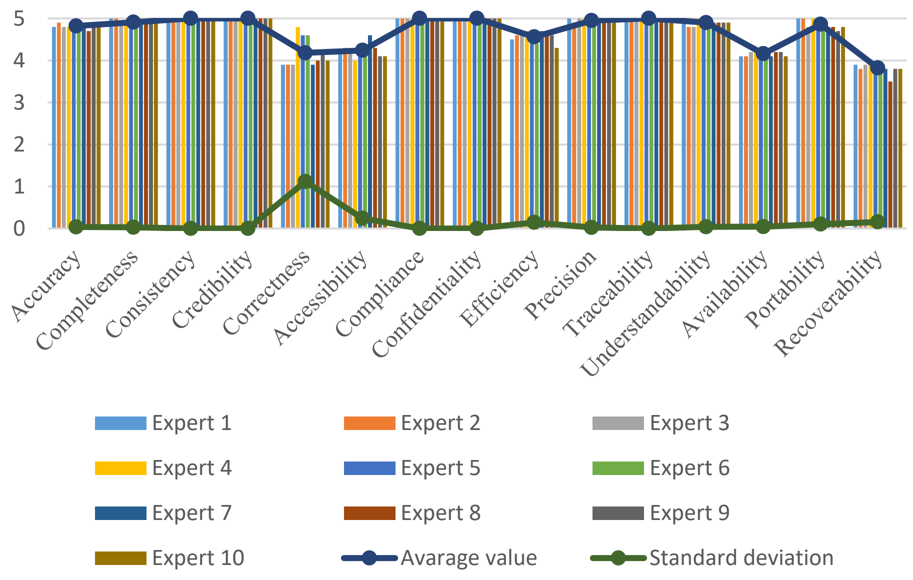


Fig. 4.14. Evaluation results of microservice

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 a database-supported consistency method, the business layer of microservice ensures that consumers operate only with consistent data models. Consumers using 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 the 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 readability. The AQL query language used to query polyglot persistence is considered

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

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 datacentre-to-datacentre 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 summarised 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.

4.6. Discussions

This chapter provides the results of the comparison between the author’s proposed monolith database migration approach and the alternative technique for extracting microservices from monolith enterprise systems. The author has chosen to compare its approach with a technique proposed by Levcovitz et al. (2016) because methods proposed by other authors do not provide or provide very little detail on how to adopt data storage to microservice architecture during the migration from monolith to microservice architecture. The advantages and disadvantages of the author’s proposed approach compared with the alternative proposed technique are shown in Table 4.5. The sign “+” means that the criterion is an advantage, while the sign “–” means that the criterion is a disadvantage or there is no mention of this criterion. The final grades were based on common agreements between the authors of the research.

Table 4.5. 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	Micro-service
1.	Possible improvement of the quality of consistency, understandability, availability, and portability	+	–
2.	Availability to use different data models for different data structures	+	–
3.	Database adaptation to microservice architecture	+	–
4.	Extensive involvement of business experts in the migration process	–	+
5.	Ability to divide database per microservice	–	+

Three advantages of the migration approach proposed in the dissertation were identified. First, it allowed us to improve the quality of consistency, understandability, availability, and portability, while the technique proposed by Levcovitz et al. 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 utilises the advantages of the microservice architecture. Meanwhile, the alternative technique divides the monolith database by tables and reuses the same legacy relational data store. Third, it allows for the extraction of the database from the monolith application and its adaptation 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 for the migration to be conducted gradually and to combine other migration methods for code decomposition.

Two disadvantages of the proposed migration approach were identified as well. First, extensive involvement of business experts is required 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, the technique proposed by Levcovitz et al. allows for the division of the database per microservice, while the method proposed in the dissertation extracts the database and converts it to the microservice.

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 analyse 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.

4.7. Conclusions of the Fourth Chapter

This chapter of the dissertation provided an evaluation of the proposed 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. The following conclusions have been drawn:

1. 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.
2. By using the proposed migration approach, it is possible to improve the quality of the consistency, understandability, availability, and portability attributes.
3. Three advantages of the proposed migration approach were identified compared to the technique proposed by Levcovitz et al.: quality improvement of consistency, understandability, availability, and portability quality attribute, the use of multi-model polyglot persistence, which allows for better utilisation of the advantages of the microservice architecture, and gradual migration and the combined use of other migration methods for code decomposition.

General Conclusions

1. The performed literature review has shown that microservice architecture is becoming the de facto industry standard for building new enterprise applications. To remain competitive, companies have started to modernise their legacy monolithic systems by decomposing them into microservices. However, the migration from a monolithic architecture to a microservice architecture is a complex challenge, which consists of issues such as microservices identification, code decomposition, a combination between microservices, independent deployment, etc. Each enterprise application is unique. It was programmed using different programming languages and techniques, and different databases and communication mechanisms were used; therefore, it creates different challenges. Although the topic of monolithic software migration into microservice architecture has already been explored by scientists and software engineers, it is a complex and relatively new challenge; therefore, there is still little research on many parts of it, such as database adaptation during the migration, communication establishment between microservices. The primary focus of most of the research is microservice identification within monolith applications and source code decomposition into microservices.

2. To address the prevailing deficiencies in communication and database components, a novel migration approach grounded in experimental investigations has been developed. This approach encompasses three primary elements: code decomposition techniques, communication establishment, and database adaptation. The innovative evaluation criteria and guidelines, derived from empirical findings, serve to recommend the most suitable code decomposition method and communication technology, considering their respective merits and demerits. To facilitate the transition of the database to a microservice architecture, a novel approach employing multi-model polyglot persistence has been proposed and assessed through experimental evaluation.
3. Three code decomposition methods were chosen for analysis and comparison: *Code-based*, *Business domain-based*, *and Storage-based*. The comparison between selected methodologies was done by decomposing the same enterprise legacy monolith application into microservices using all three selected methodologies.
   3.1. The number of extracted microservices and the size of each microservices mostly depend on the chosen microservice responsibility. There are two types of responsibilities: *business domain* and *technical function*. Microservices based on *technical function* provide higher granularity.
   3.2. The *Business-domain-based method* or the *Code-based method with semantic coupling strategy* methods are recommended for decomposing legacy monolith applications into microservices based on business domains.
   3.3. *Storage-based methods* or *Code-based methods with logical coupling strategy* methods are recommended for decomposing legacy monolith applications into microservices based on functions.
4. Five communication technologies, such as HTTP Rest, RabbitMQ, Kafka, gRPC, and GraphQL, have been evaluated and compared by the proposed evaluation criteria. The advantages and disadvantages of each communication technology were identified in the context of microservices architecture.
   4.1. If latency and throughput are the main criteria during the transition from a monolith architecture to a microservice architecture, then RabbitMQ and gRPC are the most suitable technologies. RabbitMQ showed the best results in RPC latency and throughput tests for small messages (up to 0.1MB and data model up to 100 properties), while gRPC showed the best results in RPC latency and throughput tests for big messages.

4.2. Kafka and RabbitMQ showed the best throughput results in the most loaded conditions: requested by more than 100 clients at the same time and processing 1,000,000 characters of messages. However, the latency of RPC was high, more than one second.

4.3. With the smallest request and response message size, HTTP Rest is the recommended communication technology when message size is a crucial criterion for selection.

4.4. Given its minimal storage requirements, the gRPC library is the correct choice for microservices operating in environments with limited storage capacity.

4.5. As RabbitMQ and Kafka utilise the least amount of memory, they are the recommended choices for implementation when memory size is a critical criterion.

5. The monolith database migration to a multi-model polyglot persistence based on microservices was proposed, executed as a proof-of-concept, and evaluated by domain and IT experts. Fleiss' kappa κ inter-rater agreement was used to assess the agreement among the experts (Fleiss et al., 2003). The coefficient value was 0.77, which indicates a relatively high level of agreement between the experts. The research results showed that the proposed approach could be used to conduct data storage migration from a monolith to a microservice architecture and improve the quality of the consistency, understandability, availability, and portability attributes. Moreover, it is expected that research results could inspire researchers and practitioners toward further work aimed at improving and automating the proposed approach.

References

Al–Debagy, O., & Martinek, P. (2018). A comparative review of microservices and monolithic architectures. In *Proceedings of IEEE 18th International Symposium on Computational Intelligence and Informatics – CINTI* (pp. 149–154). https://doi.org/10.1109/CINTI.2018.8928192

Anand, M. (2021). *Microservices and the Data Layer – a New IDC InfoBrief*. https://redis.com/blog/microservices–and–the–data–layer–new–idc–infobrief/

Andrawos, M. (2018). *Modern cloud native architecture: What you need to know about micro–services, containers and serverless*. http://superuser.openstack.org/articles/modern–cloud–native–architecture–what–you–need–to–know–about–micro–services–containers–and–serverless/

ArrangoDB. (2023). *ArangoDB*. https://www.arangodb.com

Atchison, L. (2018). *Microservice Architectures: What They Are and Why You Should Use Them*. https://blog.newrelic.com/technology/microservices–what–they–are–why–to–use–them/

Azarny, I. (2017). *CI/CD for Containerized Microservices*. https://dzone.com/articles/cicd–for–containerised–microservices

Azevedo, L. G., Ferreira RD, S., Silva VT, D., de Bayser, M., Soares, E. F. D. S., & Thiago, R. M. (2019). Geological Data Access on a Polyglot Database Using a Service Architecture. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures* (pp. 103–112). https://doi.org/10.1145/3357141.3357603

Azevedo, L. G., Ferreira, R. S., Silva, V. T., Bayser, M., Soares, E. F. de S., & Thiago, R. M. (2019). Geological Data Access on a Polyglot Database Using a Service Architecture. In *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse* (pp. 103–112). https://doi.org/10.1145/3357141.3357603

Balalaie, O., Heydarnoori, A., & Jamshidi, P. (2016). Microservice architecture enables DevOps. *Journal of IEEE Software*, *33*(3), 42–52. https://doi.org/10.1109/MS.2016.64

Bandhamneni, N. (2018). *Inter-service communication in Microservices*. https://walkingtreetech.medium.com/inter-service-communication-in-microservices-c54f41678998

Banijamali, A., Kuvaja, P., Oivo, M., & Jamshidi, P. (2020). Kuksa: Self–adaptive microservices in automotive systems in Product–Focused Software Process Improvement. In *International Conference on Product-Focused Software Process Improvement* (pp. 367–384). Springer, Cham. https://doi.org/10.1007/978-3-030-64148-1_23

Benchmarkdotnet community. (2023). *Benchmarkdotnet*. https://benchmarkdotnet.org/articles/overview.html

Beni, E. H., Lagaisse, B., & Joosen, W. (2019). Infracomposer: Policy–driven adaptive and reflective middleware for the cloudification of simulation & optimization workflows. *Journal of Systems Architecture*, 95, 36–46. https://doi.org/10.1016/j.sysarc.2019.03.001

Biswas, R., Xiaoyi, L., & Panda D. K. (2018). Designing a Micro–Benchmark Suite to Evaluate gRPC for TensorFlow: Early Experiences. In *Proceedings of The Ninth Workshop on Big Data Benchmarks, Performance, Optimization and Emerging Hardware*. https://doi.org/10.48550/arXiv.1804.01138

Blanch, R. (2017). *Microservices: Strategies for Migration in a Brownfield Environment*. https://medium.com/@rhettblanch_48135/microservices-strategies-for-migration-in-a-brownfield-environment-6c14335a8069

Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *Journal of IEEE Access*, 10, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

Brewer, E. A. (2000). Towards robust distributed systems. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*. https://doi.org/10.1145/343477.343502

Brito, G., & Valente, M. (2020). Microservices. REST vs GraphQL: A Controlled Experiment. In *Proceedings of 2020 IEEE International Conference on Software Architecture* (pp. 81–91). https://doi.org/10.1109/ICSA47634.2020.00016

Brown, K., & Bobby, W. (2016). Implementation patterns for microservices architectures. In *Proceedings of the Pattern Language of Programs Conference* (pp. 1–35), Allerton Park.

Carrasco, A., Bladel, B. V., & Demeyer, S. (2018). Migrating towards Microservices: Migration and Architecture Smells. In *Proceedings of the 2nd International Workshop on Refactoring* (pp. 1–6). https://doi.org/0.1145/3242163.3242164

Carrasco, A., Bladel, B., & Demeyer, S. (2018). Migrating towards microservices: Migration and architecture smells. In *Proceedings of the 2nd International Workshop on Refactoring* (pp. 1–6). https://doi.org/10.1145/3242163.3242164

Carvalho, L., Garcia, A., Assunção, W., Mello, R., & de Lima, M.J. (2019). Analysis of the criteria adopted in industry to extract micro–services. In *Proceedings of the 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Indus–try (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice* (pp. 22–29). https://doi.org/10.1109/CESSER–IP.2019.00012

Cerny, T., Donahoo, M., & Trnka, M. (2018). Contextual understanding of microservice architecture: current and future directions. *Journal of ACM SIGAPP Applied Computing Review*, 17, 29–45. https://doi.org/10.1145/3183628.3183631

Chawla, H., & Kathuria, H. (2019) *Building Microservices Applications on Microsoft Azure*. Apress.

Chen, R., Li, S., & Li, Z. (2017). From Monolith to Microservices: A Dataflow–Driven Approach. In *Proceedings of 24th Asia-Pacific Software Engineering Conference – APSEC* (pp. 466–475). https://doi.org/10.1109/APSEC.2017.53

Columbus, L. (2019). *IDC Top 10 Predictions for Worldwide IT*. https://www.forbes.com/sites/louiscolumbus/2018/11/04/idc–top–10–predictions–for–worldwide–it–2019/?sh=5e55583c7b96.

Cruz, P., Astudillo, H., Hilliard, R., & Collado, M. (2019). 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* (pp. 174–181). https://doi.org/10.1109/ICSA–C.2019.00039

Dave, A., & Degioanni, L. (2016). *The Five Principles of Monitoring Microservices*. https://thenewstack.io/five–principles–monitoring–microservices/

Davoudian, A., Chen, L., & Liu, M. (2018). A Survey on NoSQL Stores. *Journal of ACM Computing Surveys*, 51, 1–43. https://doi.org/10.1145/3158661

Dayaratna, A. (2019). *Key Considerations for Application Transformation and Modernization Using Microservices*. https://www.idc.com/getdoc.jsp?containerId=US45714619

DB-ENGINES. (2023). *DB–Engines Ranking*. https://db–engines.com/en/ranking

De Camargo, A., Salvadori, I., Mello, R. D. S., & Siqueira, F. (2016). An architecture to automate performance tests on microservices. In *Proceedings of the 18th International Conference Web–Based Applied Services* (pp. 422–429). https://doi.org/10.1145/3011141.3011179

Dehghani, Z. (2018). *How to break a Monolith into Microservices*. https://martinfowler.com/articles/break–monolith–into–microservices.html

Douglass, M. (2018). *Understanding Microservices: From Idea to Starting Line*. https://medium.freecodecamp.org/microservices–from–idea–to–starting–line–ae5317a6ff02

Esposte, A.M., Kon, F., Costa, F.M., & Lago, N. (2017). 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* (pp. 35–46). https://doi.org/10.5220/0006306200350046

Fan, C., & Ma, S. (2017). Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. In *Proceedings of the 2017 IEEE International Conference on AI & Mobile Services* (pp. 109–112). https://doi.org/10.1109/AIMS.2017.23

Fernandes, J., Lopes, I., & Rodrigues, J. (2013). Performance evaluation of RESTful web services and AMQP protocol. In *Proceedings of International Conference on Ubiquitous and Future Networks* (pp. 810–514). https://doi.org/10.1109/ICUFN.2013.6614932

Fleiss, J. L., Levin, B., & Paik, M. C. (2003). *Statistical methods for rates and proportions* (3rd ed.). John Wiley & Sons.

Fowler, M., & Lewis, J. (2014). *Microservices*. http://martinfowler.com/articles/microservices.html

Francesco, P. D., Lago, P., & Malavolta, I. (2018). Migrating towards microservice architectures: An industrial survey. In *Proceedings of the International conference on software architecture - IEEE* (pp. 29–2909). https://doi.org/10.1109/ICSA.2018.00012

Francesco, P., Malavolta, I., & Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *Proceedings of International Conference on Software Architecture* (pp. 21–30). https://doi.org/10.1109/ICSA.2017.24

Fritzsch, J., Bogner, J., Zimmermann, A., & Wagner, S. (2018). From monolith to microservices: A classification of refactoring approaches. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment* (pp. 128–141). Springer, Cham. https://doi.org/10.1007/978-3-030-06019-0_10

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., & Barros, A. (2018). Migrating Enterprise Legacy Source Code to Microservices. *Journal of IEEE Software*, 35, 63–72. https://doi.org/10.1109/MS.2017.440134612

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., & Barros, A. (2018). Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency. *Journal of IEEE Software*, 35, 63–72. https://doi.org/10.1109/MS.2017.440134612

Galbraith, K. (2019). *3 methods for microservice communication*. https://blog.logrocket.com/methods-for-microservice-communication/

Ghofrani, J., & Bozorgmehr, A. (2019). Migration to microservices: Barriers and solutions in Applied Informatics. In *Proceedings of Second International Conference - ICAI 2019* (pp. 269–281). https://doi.org/10.1007/978-3-030-32475-9_20

Gouigoux, J. P., & Tamzalit, D. (2017). From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Orient–ed Architecture. In *Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops* (pp. 62–65). https://doi.org/10.1109/ICSAW.2017.35

GraphQL. (2023). *GraphQL – A query language for your API*. https://www.graphql.org

gRPC. (2023). *gRPC – A High–Performance, Open–Source Universal RPC Framework*. https://www.grpc.io.

Hartig, O., & Perez J. (2017). *Microservices. An Initial Analysis of Facebook's GraphQL Language.* AMW.

Hasselbring, W., & Steinacker, G. (2017). Microservice Architectures for Scalability, Agility and Reliability in E–Commerce. In *Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops* (pp. 243–246). https://doi.org/10.1109/ICSAW.2017.11

Hong, X., Yang, H., & Kim, Y. (2018). Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. In *Proceedings of 2018 International Conference on Information and Communication Technology Convergence* (pp. 257–259).. https://doi.org/10.1109/ICTC.2018.8539409

Kalske, M., Mäkitalo, N., & Mikkonen, T. (2017). Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering* (pp. 32–47). Springer, Cham. https://doi.org/10.1007/978-3-319-74433-9_3

Karwowski, W., Rusek, M., Dwornicki, G., & Orłowski, A. (2018). Swarm based system for management of containerized microservices in a cloud consisting of heterogeneous servers. In *Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017* (pp. 262–271). Springer, Cham. https://doi.org/10.1007/978-3-319-67220-5_24

Khine, P. P., & Wang, Z. (2019). A Review of Polyglot Persistence in the Big Data World. *Journal of Information*, 10, 1-141. https://doi.org/10.3390/info10040141

Knoche, H., & Hasselbring, W. (2018). Using Microservices for Legacy Software Modernization. *Journal of IEEE Software,* 35, 44–49. https://doi.org/10.1109/MS.2018.2141035.

Knoche, H., & Hasselbring, W. (2019). Drivers and Barriers for Microservice Adoption—A Survey among Professionals in Germany. *Journal of Enterprise Modelling and Information Systems Architectures. (EMISAJ)–Int. J. Concept*, 14, 1–35. https://doi.org/10.18417/emisa.14.1

Koltovich, S. (2017). *How to Modernize Legacy Applications for a Microservices–Based Deployment*. https://thenewstack.io/modernize–legacy–applications–keep–update–re–write–needs–re–written/

Krishnan, G. (2002). IBM Mainframe Database Overview and Evolution of DB2 as Web Enabled Scalable Server. *Journal of Datenbank–Spektrum*, 3, 6–14.

Krylovskiy, A., Jahn, M., & Patti, E. (2015). 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* (pp. 25–30). https://doi.org/10.1109/FiCloud.2015.55

Kwiecen, A. (2019). *10 companies that implemented the microservice architecture and paved the way for others*. https://www.divante.com/blog/10–companies–that–implemented–the–microservice–architecture–and–paved–the–way–for–others

Laigner, R., Zhou, Y., Salles, M. A. V.; Liu, Y., & Kalinowski, M. (2021). Data Management in Microservices: State of the Practice, Challenges, and Research Directions. In *Proceedings of the Proceedings of the VLDB Endowment*, 14, (pp. 3348–3361). https://doi.org/10.14778/3484224.3484232

Lenarduzzi, V., & Sievi-Korte, O. (2018). On the negative impact of team independence in microservices software development. In *Proceedings of the 19th International Conference on Agile Software Development: Companion* (pp. 1–4). https://doi.org/10.1145/3234152.3234191

Lenarduzzi, V., Lomio, F., Saarimäki, N., & Taibi, D. (2020). Does migrating a monolithic system to microservices decrease the technical debt? *Journal of Systems and Software*, 169. https://doi.org/10.1016/j.jss.2020.110710

Levcovitz, A., Terra, R., & Valente, M. T. (2015). Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. In *Proceedings of the 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance* (pp. 97–104). https://doi.org/10.48550/arXiv.1605.03175

Levcovitz, A., Terra, R., & Valente, M. T. (2016). Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. In *Proceedings of 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance* (pp. 97–104). https://doi.org/10.48550/arXiv.1605.03175

Linthicum, D. (2018). *From containers to microservices: Modernizing legacy applications*. https://techbeacon.com/enterprise–it/containers–microservices–modernizing–legacy–applications

Lotz, J., Vogelsang, A., Benderius, O., & Berger, C. (2019). Microservice Architectures for Advanced Driver Assistance Systems: A Case–Study. In *Proceedings of the 2019 IEEE International Conference on Software Architecture Companion* (pp. 45–52). https://doi.org/10.1109/ICSA–C.2019.00016

Luz, W., Agilar, E., Oliveira, M. S., Melo, C. E. R., Pinto, G., & Bonifácio, R. (2018). An Experience Report on the Adoption of Micro–services in Three Brazilian Government Institutions. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering* (pp. 32–41). https://doi.org/10.1145/3266237.3266262

Mayer, B., & Weinreich, R. (2018). An Approach to Extract the Architecture of Microservice–Based Software Systems. In *Proceedings of 2018 IEEE Symposium on Service–Oriented System Engineering (SOSE)*. https://doi.org/10.1109/SOSE.2018.00012

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. In *Proceedings of 2017 IEEE International Conference on Web Services* (pp. 524–531). https://doi.org/10.1109/ICWS.2017.61.

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. In *Proceedings of the 2017 IEEE International Conference on Web Services* (pp. 524–531). https://doi.org/10.1109/ICWS.2017.61

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S. T., & Dustdar, S. (2018). Microservices: Migration of a Mission Critical System. *Journal of IEEE Transactions on Services Computing,* 14, 1464–1477. https://doi.org/10.48550/arXiv.1704.04173

Meier, A., & Kaufmann, M. (2018). SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management. Springer. https://doi.org/10.1007/978–3–658–24549–8

Melendez C., & McAllister, D. (2018). *Microservices Logging Best Practices*. https://dzone.com/articles/microservices–logging–best–practices

Messina, A., Rizzo, R., Storniolo, P., & Urso, A. (2016). 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* (pp. 223–233). https://doi.org/10.13140/RG.2.1.3529.3681

Microsoft, (2020). *Communication in a microservice architecture*. https://docs.microsoft.com/en–us/dotnet/architecture/microservices/architect–microservice–container–applications/communication–in–microservice–architecture

Microsoft. (2022). *Code metrics values*. https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022

Microsoft. (2023). *C Sharp documentation*. https://docs.microsoft.com/en–us/dotnet/csharp

Microsoft. (2023). *NuGet*. https://www.nuget.org

Microsoft. (2023). *Visual Studio*. https://visualstudio.microsoft.com

Mishra, M., Kunde, S., & Nambiar, M. (2017). Cracking the Monolith: Challenges in Data Transitioning to Cloud Native Architectures. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* (pp. 1–4). https://doi.org/10.1145/3241403.3241440

Mishra, M., Kunde, S., & Nambiar, M. (2018). Cracking the Monolith: Challenges in Data Transitioning to Cloud Native Architectures. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* (pp. 1–4). https://doi.org/10.1109/MS.2017.440134612

Mohamed, D., Mezouari, A., Faci, N., Benslimane, D., Maamar, Z., & Fazziki, A. (2021). A multi–model based microservices identification approach. *Journal of Systems Architecture,* 118. https://doi.org/10.1016/j.sysarc.2021.102200

Montesi, F., & Weber, J. (2016). Circuit Breakers, Discovery, and API Gateways in Microservices. *Journal of ArXiv,* abs/1609.05830. https://doi.org/10.48550/arXiv.1609.05830

Mulesoft, (2023). *https://www.mulesoft.com/resources/api/microservices–devops–better–together*. https://www.mulesoft.com/resources/api/microservices–devops–better–together

Nayak, A., Poriya, A., & Poojary, D. (2013). Type of NoSQL databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5, 16–19. https://doi.org/10.5120/ijais12-450888

NBomber. (2023). *NBomber*. https://nbomber.com/docs/overview/

Newman, S. (2019). *Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith*, 1st ed. O'Reilly Media.

Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., & Geiger, S. (2020). Assessing Architecture Conformance to Coupling–Related Patterns and Practices in Microservices. In *Proceedings of European Conference on Software Architecture* (pp. 3–20). L'Aquila. https://doi.org/10.1007/978–3–030–58923–3_1

Olofson, C., & Chen, G. (2021). *The Impact of Application Modernization on the Data Layer*. https://redis.com/docs/application–modernizaton–impact–on–data–layer/

Pozdniakova, O., & Mažeika, D. (2017). A cloud software isolation and crossplatform portability methods. In *Proceedings of Open Conference of Electrical, Electronic and Information Sciences - eStream* (pp. 1–6). https://doi.org/10.1109/eStream.2017.7950315.

Pozdniakova, O., & Mažeika, D. (2017). Systematic Literature Review of the Cloud–ready Software Architecture. *Journal of Modern Computing*, 5, 124–135. https://doi.org/10.22364/bjmc.2017.5.1.08

Rajasekharaiah, C. (2021). Case Study: Energence. In *Book Cloud-Based Microservices* (pp. 1–12). Apress. https://doi.org/10.1007/978-1-4842-6564-2_1

Ramin, F., Matthies, C., & Teusner, R. (2020). More than code: Contributions in scrum software engineering teams. In *Proceedings of IEEE/ACM 42nd International Conference on Software Engineering Workshops* (pp. 137–140). https://doi.org/10.1145/3387940.3392241

Richter, D., Konrad, M., Utecht, K., & Polze, A. (2017). Highly–Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. In *Proceedings of 2017 IEEE International Conference on Software Quality, Reliability and Security Companion* (pp. 130–137). https://doi.org/10.1109/TSC.2018.2889087

Rosendahl, H. (2016). *Containers vs Virtual Machines (vms) – A Security Perspective*. https://neuvector.com/container–security/containers–vs–virtual–machines–vms/

Serra, J. (2015). *What is Polyglot Persistence?* https://www.jamesserra.com/archive/2015/07/what–is–polyglot–persistence/

Shah, C., Srivastava, K., & Shekokar, N.M. (2016). 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* (pp. 40–45). https://doi.org/10.1109/IC3e.2016.8009037

Sharma, V., & Dave, M. (2012). SQL and NoSQL Databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 12, 467–471. https://doi.org/10.1145/3158661

Singhal, H., Saxena, A., Mittal, N., Dabas, C., & Kaur, P. (2021). Polyglot Persistence for Microservices–Based Applications. *Journal of Information Technologies and Systems Approach*, 14, 17–32. https://doi.org/10.4018/IJITSA.2021010102

Smid, A., Wang, R., & Cerny, T. (2019). Case study on data communication in microservice architecture. *In Proceedings of the Conference on Research in Adaptive and Convergent Systems* (pp. 261–267), Chongqing, China. https://doi.org/10.1145/3338840.3355659

Soldani, J., Tamburri, D. A., & Van Den Heuvel, W. J. (2018). The pains and gains of microservices: A Systematic grey literature review. *Journal of System Software*, 146, 215–232. https://doi.org/10.1016/j.jss.2018.09.082

Soroko, A. (2017). *Cloud Foundry Deployment Metrics That Matter Most*. https://www.altoros.com/blog/cloud–foundry–deployment–metrics–that–matter–most/

Štefanko, M., Chaloupka, O., & Rossi, B. (2019). The saga pattern in a reactive microservices environment. In *Proceedings of International Conference on Software and Data Technologies* (pp. 483–490). https://doi.org/10.5220/0007918704830490

Taibi, D., Lenarduzzi, V., & Pahl, C. (2020). Architectural Patterns for Microservices: A Systematic Mapping Study. In *Proceedings of International Conference on Cloud Computing and Services Science* (pp.100–104). https://doi.org/10.5220/0006798302210232

Terzic, B., & Dimitrieski, V. (2018). A model–driven approach to microservice software architecture establishment. In *Proceedings of 2018 Federated Conference on Computer Science and Information Systems* (pp. 73 –80). https://doi.org/10.15439/2018F370

Trivedi, K., Shah, S., & Srivastava, K. (2018). An Efficient E–Commerce Design by Implementing a Novel Data Mapper for Polyglot Persistence. In *Proceedings of 2nd International Conference on Advanced Computing Technologies and Applications - ICACTA 2020* (pp. 149–156). https://doi.org/10.1007/978-981-15-3242-9_15

Vennaro, N. (2017). *How to introduce microservices in a legacy environment*. https://www.infoworld.com/article/3237175/how–to–introduce–microservices–in–a–legacy–environment.html

Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., & Nieh, J. (2015). Synapse: A microservices architecture for heterogeneous–database web applications. In *Proceedings of the 10th European Conference on Computer Systems* (pp. 1–16). https://doi.org/10.1145/2741948.2741975

Villaça, L. H., Azevedo, L. G., & Siqueira, S. W. (2020). Microservice Architecture for Multistore Database Using Canonical Data Model. In *Proceedings of the XVI Brazilian Symposium on Information Systems, São Bernardo do Campo* (pp. 1–8). https://doi.org/10.1145/3411564.3411629

Walsh, K., & Manferdelli, J. (2017). Mechanisms for Mutual Attested Microservice Communication. In *Proceedings of the 10th International Conference on Utility and Cloud Computing* (pp. 59–64). https://doi.org/10.1145/3147234.3148102

Wang, Y., Kadyala, H., & Rubin, J. (2020). Promises and Challenges of Microservices: An Exploratory Study. *Journal of Empirical Software*, *26*(4), 1–44. https://doi.org/10.1007/s10664–020–09910–y.

Wiese, L. (2015). Polyglot Database Architectures. In *Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB*, Trier.

Wireshark. (2023). *Wireshark*. https://www.wireshark.org/

Wolfart, D., Assunção, W., Silva, I., Domingos, D., Schmeing, E., Villaca, G., & Paza, D. (2021). Modernizing Legacy Systems with Microservices: A Roadmap. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering* (pp. 149–159). https://doi.org/10.1145/3463274.3463334

Yarygina, T., & Bagge, A. (2018). Overcoming Security Challenges in Microservice Architectures. In *Proceedings of 2018 IEEE Symposium on Service–Oriented System Engineering* (pp. 11–20). https://doi.org/10.1109/SOSE.2018.00011

Zdepski, C., Bini, T. A., & Matos, S. N. (2018). An Approach for Modelling Polyglot Persistence. In *Proceedings of the International Conference on Information Systems (ICEIS)*. https://doi.org/10.5220/0006684901200126

Zdepski, C., Bini, T. A., & Matos, S. N. (2020). PDDM: A Database Design Method for Polyglot Persistence. *American Academic Scientific Research Journal for Engineering, Technology, and Sciences*, *71*(1), 136–152.

List of Scientific Publications by the Author on the Topic of the Dissertation

Papers in the Reviewed Scientific Journals

Kazanavicius, J., Mazeika, D., & Kalibatiene, D. (2022). An Approach to Migrate a Monolith Database into Multi–Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database. *Journal of Applied Sciences*, *12*(12), 6189. https://doi.org/10.3390/app12126189

Kazanavicius, J., & Mazeika, D. (2023). The Evaluation of Microservice Communication While Decomposing Monoliths. *Journal of Computing and Informatics*, *42*(1), 1–36. https://doi.org/10.31577/cai_2023_1_1

Papers in Other Editions

Kazanavicius, J., & Mazeika, D. (2019). Migrating Legacy Software to Microservices Architecture. In *Proceedings of the 2019 Open Conference of Electrical, Electronic and Information Sciences – eStream* (pp. 1–5). https://doi.org/10.1109/estream.2019.8732170

Kazanavicius, J., & Mazeika, D. (2020). Analysis of Legacy Monolithic Software Decomposition into Microservices. In *Proceedings of the Baltic–DB&IS–Forum–DC 2020* (pp. 25–32). https://ceur-ws.org/Vol-2620/paper4.pdf

Kazanavicius, J., & Mazeika, D. (2023). An Approach to Migrate Legacy Monolithic Application in Microservice Architecture. In *Proceedings of the 2023 Open Conference of Electrical, Electronic and Information Sciences – eStream* (pp. 1–6). https://doi.org/10.1109/eStream59056.2023.10135021

Summary in Lithuanian

Įvadas

Problemos formulavimas

Atsižvelgiant į daugybę pastaraisiais metais sėkmingai įgyvendintų projektų, naudojant mikroservisų architektūrą, ji tapo standartu, pagal numatytuosius parametrus daugumoje įmonių kuriant naują ir modernizuojant jau esamą programinę įrangą. Didžiosios įmonės, tokios kaip „Amazon“, „EBay“, „Netflix“, „PayPal“, „Twitter“ ir kitos, sėkmingai perėjo nuo monolitinės architektūros prie mikroservisų architektūros.

Mikroservisų architektūrą kombinuojant kartu su programinės įrangos kūrimo ir IT operacijų (DevOps) praktika, pagerinamas programinės įrangos kūrimo judrumas ir lankstumas. Įmonės gali greičiau pristatyti savo skaitmeninius produktus ir paslaugas labai konkurencingai rinkai. Mikroservisų architektūra tampa šiuolaikinių debesų kompiuterijos pagrindu veikiančių programinės įrangos sistemų projektavimo standartu, nes ji geriausia išnaudoja debesų kompiuterijos privalumus. Kartu naudojant mikroservisų architektūros ir debesų technologijas, sutrumpinamas programinės įrangos kūrimo laikas ir padidinamas diegimo greitis.

Perkėlimas iš monolitinės architektūros į mikroservisų architektūrą yra sudėtingas kompleksinis iššūkis, apimantis tokias problemas kaip mikroservisų identifikavimą, išeities kodo išskaidymą, ryšio tarp mikroservisų užmezgimą, duomenų bazės adaptaciją, nepriklausomą diegimą ir kt. Mikroservisų identifikavimas ir išskyrimas iš esamos monolitinės programinės įrangos yra labai sudėtinga užduotis. Pažymėtina, kad kiekviena įmonės programa yra unikali, nes buvo programuojama naudojant skirtingas programavimo kalbas ir technikas, skirtingas duomenų bazės ir komunikacijos technologijas. Atsižvelgiant į tai kiekviena monolitinė programa kuria skirtingus iššūkius. Skirtingos organizacijos taiko skirtingus perkėlimo modelius ir metodus, nes mikroservisų architektūra vis dar yra palyginti naujas architektūrinis požiūris, o plačiai patvirtinto būdo, kaip atlikti perkėlimą iš monolitinės programos, nėra.

Darbo aktualumas

Tarptautinės duomenų korporacijos duomenimis, 89 % iš maždaug 300 Šiaurės Amerikos įmonių apklausos respondentų jau naudoja mikroservisų architektūrą kuriant programinę įrangą (Olofson et al., 2021; Anand, 2021). Tarptautinė duomenų korporacija prognozuoja, kad 90 % visų naujų programų bus sukurtos remiantis mikroservisų architektūra. Įmonės, siekdamos išlikti konkurencingos rinkoje, pradėjo modernizuoti savo esamas monolitines sistemas, išskaidydamos jas į mikroservisus (Francesco et al., 2018; Knoche et al., 2018; Wang et al., 2020; Wolfart et al.)., 2021; Beni et al., 2019; Mohamed et al., 2021).

Nors monolitinės programinės įrangos perkėlimo į mikroservisų architektūrą tema yra nagrinėjama mokslininkų ir programinės įrangos inžinierių, tai vis dar palyginti naujas iššūkis. Mikroservisų identifikavimas ir išeities kodo skaidymas yra plačiai išnagrinėtas, tačiau tokios temos kaip ryšio technologijos parinkimas ir ryšio užmezgimas tarp mikroservisų ar duomenų bazės adaptacija prie mikroservisų architektūros yra mažai tyrinėtos. Siekiant užpildyti šią spragą, šiame darbe pasiūlytas perkėlimo metodas, sudarytas iš trijų pagrindinių dalių: mikroservisų identifikavimo ir išeities kodo išskaidymo metodų, komunikacijos technologijos parinkimo ir duomenų bazės adaptacijos.

Tyrimo objektas

Disertacinių tyrimų objektas – taikomųjų monolitinių programų perkėlimo į mikroservisų architektūrą metodai.

Darbo tikslas

Disertacijos tikslas – pagerinti perkėlimą iš taikomųjų monolitinių programų prie mikroservisų architektūros, pasiūlant naują perkėlimo metodą, kuris apima išeities kodo išskaidymą, ryšio užmezgimą ir duomenų bazės adaptaciją.

Darbo uždaviniai

Darbo tikslui pasiekti buvo keliami šie uždaviniai:

1. Apžvelgti mikroservisų architektūros ypatumus ir esamas monolitinės programinės įrangos perkėlimo į mikroservisų architektūrą metodikas, nustatant svarbiausius aspektus bei esamas spragas.
2. Ištirti monolitinės programinės įrangos išeities kodų išskaidymo būdus migruojant į mikroservisų architektūrą.
3. Ištirti mikroservisų komunikacijos technologijas ir nustatyti konkrečius jų panaudojimo atvejus.

4. Įvertinti ir pasiūlyti monolitinės duomenų bazės perkėlimą į mikroservisų architektūrą, pagrįstą daugiamodeliniu poliglotų modeliu.
5. Pasiūlyti naują metodą perėjimui iš monolitinės architektūros prie mikroservisų architektūros, sujungiantį išeities kodo dekompoziciją, ryšio užmezgimą tarp mikroservisų ir duomenų bazės adaptavimą mikroservisų architektūrai.

Tyrimų metodika

Nagrinėjant darbo objektą, taikyti šie metodai:

1. Atlikta sisteminė mokslinės literatūros apžvalga apie esamus monolitinės programinės įrangos perkėlimo į mikroservisų architektūrą metodus. Apibendrintos kiekvieno metodo privalumai ir trūkumai. Nustatytos spragos komunikacijos ir duomenų bazių srityse.
2. Eksperimentinis tyrimo metodas, pritaikytas tiriant mikroservisų architektūros komunikacijos technologijas. Apibendrinti kiekvienos technologijos pranašumai ir trūkumai bei nustatyti konkretūs jų panaudojimo atvejai. Visi mikroservisai buvo parašyti naudojant C# programavimo kalbą. Delsos testai buvo atlikti naudojant „BenchmarkDotNet“ biblioteką. Pralaidumo testai buvo atlikti naudojant „NBomber“ biblioteką.
3. Konstruktyvus tyrimo metodas buvo pritaikytas kuriant ir patvirtinant siūlomą monolitinės duomenų bazės perkėlimo į mikroservisų architektūrą metodą. Daugiamodelinis poliglotinis modelis buvo įgyvendintas ArangoDB duomenų bazėje ir inkapsuliuotas mikroservise, parašytame C# programavimo kalba.

Darbo mokslinis naujumas

1. Siūlomas perkėlimo iš monolitinės programinės įrangos į mikroservisų architektūrą metodas išsiskiria, nes unikaliai apima tris esminius komponentus: išeities kodo dekompoziciją, ryšio užmezgimą tarp mikroservisų ir duomenų bazės adaptaciją. Esami metodai dažnai suteikia ribotą aprėptį, sprendžiant tik išeities kodo dekompozicijos problemą.
2. Pasiūlytas naujas monolitinių duomenų bazių perkėlimo į mikroservisų architektūrą metodas. Perkėlimo metu esamas duomenų modelis yra transformuojamas į daugiamodelinį poliglotinį modelį. Ši transformacija pagerina nuoseklumą, suprantamumą, prieinamumą ir perkeliamumą, kartu sėkmingai išsaugant duomenų kokybę vienuolikoje ISO/IEC 25012:2008 standarto atributų.
3. Pasiūlyti nauji išeities kodų dekompozicijos metodų ir komunikacijos technologijų vertinimo kriterijai yra pagrįsti išsamia jų privalumų ir trūkumų analize. Kriterijai suteikia novatorišką pagrindą pasirinkti vieną iš trijų kodo dekompozicijos metodų ir penkių komunikacijos technologijų, įvertintų ir palygintų pagal aštuonis kriterijus.

Darbo rezultatų praktinė reikšmė

Pasiūlytas naujas perkėlimo metodas iš esamos monolitinės programinės įrangos į mikroservisų architektūrą leidžia atlikti perkėlimą remiantis trimis pagrindiniais aspektais: išeities kodo išskaidymu, ryšio užmezgimu ir duomenų bazės transformavimu. Taikydami siūlomą perkėlimo metodą, perkėlimo vykdytojai, atsižvelgdami į savo poreikius, gali pasirinkti vieną iš trijų kodų skaidymo būdų ir vieną iš penkių komunikacijos technologijų. Tyrimo rezultatai parodė, kad siūlomas duomenų bazės perkėlimo metodas gali būti taikomas duomenų bazių perkėlimui iš monolitinės į mikroservisų architektūrą ir nuoseklumo, suprantamumo, prieinamumo ir perkeliamumo atributų kokybei pagerinti. Be to, autorius tikisi, kad darbo rezultatai gali paskatinti tyrėjus ir praktikus tolesniam darbui, siekiant pagerinti ir automatizuoti siūlomą metodą.

Ginamieji teiginiai

1. Pasiūlytas perkėlimo metodas leidžia atlikti perkėlimą iš monolitinės duomenų bazės į mikroservisų architektūrai pritaikytą daugiamodelinę poliglotinę duomenų bazę, neprarandant duomenų modelio kokybės vienuolikoje iš penkiolikos ISO/IEC 25012:2008 standarto kokybės atributų, bei pagerinti nuoseklumą, suprantamumą, prieinamumą ir perkeliamumą.
2. RabbitMQ ir gRPC yra tinkamiausios technologijos, jei delsa ir pralaidumas yra pagrindiniai komunikacijos technologijos pasirinkimo kriterijai migruojant iš monolitinės architektūros į mikroservisų architektūrą. GRPC naudojamas dvejetainis serializavimas pranoksta RabbitMQ perduodant sudėtingesnius pranešimus.
3. Kodo ir duomenų bazių elementais pagrįsti mikroservisų identifikavimo metodai leidžia identifikuoti monolitinės programos technines funkcijas ir priskirti joms atitinkamus kodo ir duomenų bazių komponentus, o verslo domenais pagrįsti mikroservisų identifikavimo metodai leidžia identifikuoti mikroservisus pagal identifikuotas verslo sritis. Mikroservisai, pagrįsti techninėmis funkcijomis, užtikrina didesnį detalumą.

Darbo rezultatų aprobavimas

Disertacijos tema paskelbta 2 žurnaluose, įtrauktuose į *Clarivate Analytics* (buv. *Thomson Reuters*) *Web of Science* duomenų bazę ir turinčiuose citavimo rodiklį, 2 – mokslinių konferencijų pranešimų rinkiniuose. Moksliniai rezultatai buvo pristatyti 4 mokslinėse konferencijose:

– *2019 Open Conference of Electrical, Electronic and Information Sciences* (eStream), 2019 m. balandžio 1 d., Vilnius, Lietuva.
– *Baltic DB&IS 2020, 14th International Baltic Conference on Databases and Information Systems*, 2020 m. birželio 16–19 d., Talinas, Estija.
– *Data Analysis Methods for Software Systems* (DAMSS), 2021 m. gruodžio 2–4 d., Druskininkai, Lietuva.
– *2023 Open Conference of Electrical, Electronic and Information Sciences* (eStream), 2023 m. balandžio 27 d., Vilnius, Lietuva.

Disertacijos struktūra

Disertaciją sudaro įvadas, penki pagrindiniai skyriai, bendrosios išvados, literatūros šaltinių sąrašas, disertacijos autoriaus publikacijų sąrašas ir santrauka lietuvių kalba. Disertacijos apimtis: 162 puslapiai, 1 formulė, 74 paveikslai ir 21 lentelė.

1. Mikroservisų architektūros ir esamų perkėlimo iš monolitinės programinės įrangos į mikroservisų architektūrą metodų analizė

Šiame skyriuje apžvelgiama mikroservisų architektūra ir jos pranašumai bei trūkumai lyginant su monolitine architektūra. Pirmiausia paaiškinami svarbiausi mikroservisų architektūros aspektai ir priežastys, kodėl įmonės siekia perkelti savo esamą monolitinę programinę įrangą. Toliau tekste pateikiama perkėlimo iš monolitinės programinės įrangos į mikroservisų architektūrą metodų analizė. Nagrinėjami įvairūs perkėlimo metodai, pateikiami jų privalumai ir trūkumai. Toliau tekste apžvelgiamos įvairios komunikacijų technologijos ir būdai, tinkami mikroservisų architektūrai. Galiausiai pateikiami duomenų bazės adaptavimo mikroservisų architektūrai literatūros analizės rezultatai.

Monolitinė architektūra yra tradicinis programinės įrangos kūrimo būdas, kai visos funkcijos yra įtrauktos į vieną programą – vientisą autonominį vienetą. Monolitinės architektūros trūkumai yra šie: labai sunku atlikti pakeitimus, kai monolitinė programa yra labai didelė ir sudėtinga, su kiekvienu atnaujinimu turi būti atnaujinta visa programa, bet kurio komponento klaida gali sugadinti visą programą (Dehghani et al., 2018; Fritzsch et al., 2018; Kalske et al., 2017). Klaidų taisymas ir naujų funkcijų įtraukimas į tokią programą yra labai sudėtingas ir daug laiko bei resursų reikalaujantis darbas. Dėl šių monolitinės architektūros trūkumų organizacijos pradeda ieškoti naujo architektūrinio sprendimo (Dehghani et al., 2018). Dėl daugybės pastaraisiais metais sėkmingai įgyvendintų projektų, naudojant mikroservisų architektūrą, ši tapo standartu pagal numatytuosius parametrus daugumoje įmonių kuriant naują ir modernizuojant esamą programinę įrangą (Kwiecen, 2019).

Mikroservisų architektūra – tai būdas sukurti vieną programą kaip mažų programėlių rinkinį, kur kiekviena programėlė veikia atskirai ir palaiko ryšį su kitomis programėlėmis lengvomis komunikacijos technologijomis, tokiomis kaip HTTP. Šios programėlės yra sukurtos remiantis atskiromis verslo sritimis ir yra nepriklausomai įdiegiamos visiškai automatizuotais diegimo mechanizmais. Šių programėlių, kurios gali būti parašytos skirtingomis programavimo kalbomis ir naudojamos skirtingos duomenų saugojimo technologijos, centralizuotas valdymas yra minimalus (Fowler et al., 2014). Pagrindiniai trys mikroservisų architektūros principai yra šie: mikroservisas turi vieną atsakomybę, mikroservisas yra autonomiškas, mikroservisas yra poliglotas (Blinowski et al., 2022).

Perkėlimas iš monolitinės architektūros į mikroservisų architektūrą yra sudėtingas ir kompleksinis iššūkis, kurį sudaro tokios problemos kaip mikroservisų identifikavimas, išeities kodo išskaidymas, mikroservisų komunikacijos užmezgimas, nepriklausomas diegimas ir kt. Vienas iš pagrindinių iššūkių šiame kontekste yra mikroservisų identifikavimas monolitinių kodų bazėse (Carrasco et al., 2018; Mazlami et al., 2017; Furda et al., 2018; Mishra et al., 2018; Linthicum, 2018). Kitas didelis iššūkis yra apibrėžti tinkamą komunikacijos technologiją. Monolitinėse programose ryšys tarp komponentų vykdomas taikant proceso metodus arba funkcijų iškvietimus. Mikroservisų architektūra pagrįsta programa yra paskirstyta sistema, veikianti keliuose procesuose ar taikant kelias paslaugas. Todėl mikroservisai turi sąveikauti naudodami tarp procesines komunikacijos technologijas (Microsoft, 2020; Cerny et al., 2018; Smid et al., 2019). Trečiasis iššūkis, duomenų bazės pritaikymas mikroservisų architektūrai, yra pripažįstamas kaip vienas iš opiausių ir viena mažiausiai nagrinėtų temų perkėlimas iš monolitinės į mikroservisų architektūrą kontekste (Laigner et al., 2021; Azevedo et al., 2019; Richter et al., 2017; Francesco et al., 2017; Knoche et al., 2019; Luz et al., 2018; Soldani et al., 2018).

Literatūros apžvalgos ir analizės metu buvo nustatytos trys pagrindinės kryptys, kaip būtų galima realizuoti perkėlimą iš monolitinės į mikroservisų architektūrą: *Duomenų bazės elementais pagrįsta kryptis* – išeities kodas, susijęs su konkrečiais duomenų bazės elementais, pavyzdžiui, duomenų bazės lentele, turi būti pateikiamas viename mikroservise. *Kodo elementais pagrįsta kryptis* – programų išskaidymas į mikroservisus turėtų būti įgyvendintas remiantis išeities kodo elementais, tokiais kaip klasė ar metodas. Mikroservisų funkcijos turėtų būti identifikuotos ir visi atitinkami kodo elementai priskirti vienai iš šių funkcijų. *Verslo domenu pagrįsta kryptis* – programa turi būti suskirstyta į mikroservisus pagal identifikuotas verslo sritis, kiekvienam verslo domenui turi būti atskiras mikroservisas (Levcovitz et al., 2016; Mazlami et al., 2017; Fan et al., 2017; Chen et al., 2017; Knoche et al., 2018).

Perkėlimo rezultatai, pasitelkiant skirtingas metodikas, buvo įvertinti ir palyginti taikant įvarius kriterijus. *Mikroservisų kandidatų skaičius* ir *mikroserviso dydis* – kriterijai, nurodantys, kokio dydžio ir kiek mikroservisų kandidatų galima potencialiai išgauti taikant pasirinktą metodiką. *Duomenų bazių* kriterijumi įvertinama, ar metodikos gali išskaidyti duomenų bazes monolitų skaidymo procese. *Mikroservisų komunikacijos* kriterijumi analizuojamas mikroservisų kaip vieno sprendimo veikimas po dekomponavimo proceso. *Automatizavimo* kriterijumi įvertinamos kiekvienos metodikos galimybės būti visiškai automatizuotoms. Analizuojant kriterijus *technologijos* ir *įrankiai,* pateikiama daugiau informacijos apie tai, kaip būtų galima realizuoti metodikas ir kokias technologijas bei priemones būtų galima taikyti. Paskutinis kriterijus, *kodo kokybė,* įvertina kodo kokybės poveikį perkėlimo procese.

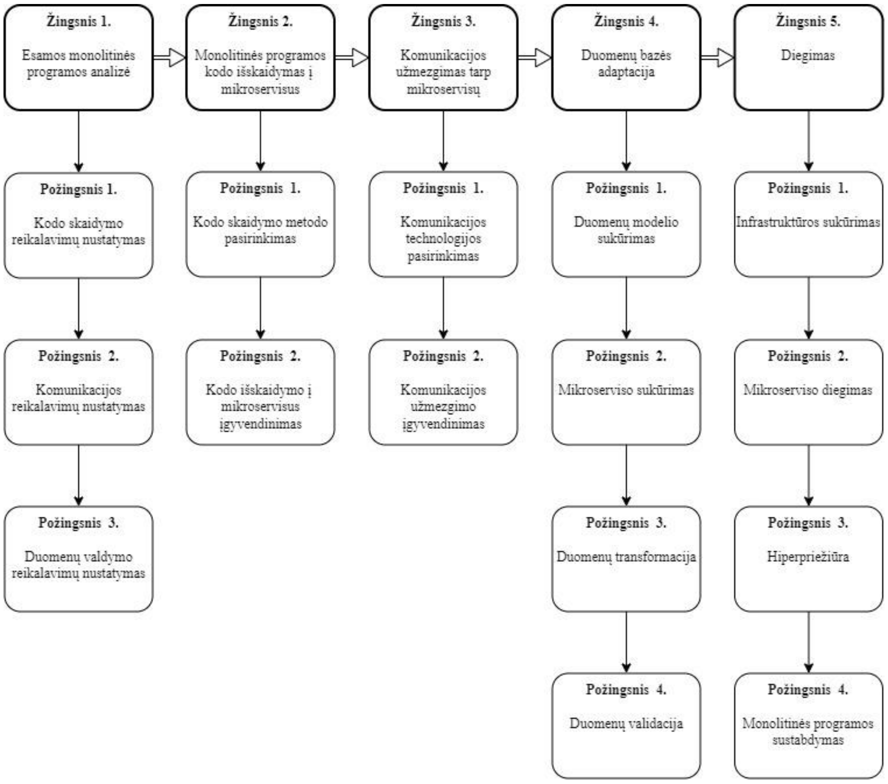
Kiekviena monolitinė programa yra unikali, sukurianti unikalius iššūkius. Naudojamos technologijos sudėtingumas, verslo domenas, komandos dydis ar jos įgūdžiai – tai parametrai, kurie kiekvienu atveju gali būti labai skirtingi. Kiekvienas atvejis yra skirtingas ir organizacija turėtų pasirinkti, kuris metodas ar metodų rinkinys geriausiai tinka perkėlimui iš monolitinės į mikroservisų architektūrą. Pasirinkta metodika arba metodikų rinkinys turėtų turėti galimybę išgauti mikroservisus pagal pasirinktus kriterijus ir būti suderinami su įmonės naudojamomis technologijomis. Kodo ir duomenų bazių elementais pagrįsti mikroservisų identifikavimo metodai leidžia identifikuoti monolitinės programos technines funkcijas ir priskirti joms atitinkamus kodo ir duomenų bazių komponentus, o verslo domenais pagrįsti mikroservisų identifikavimo metodai leidžia identifikuoti mikroservisus pagal identifikuotus verslo domenus. Mikroservisai, pagrįsti techninėmis funkcijomis, užtikrina didesnį detalumą. Nė viena iš analizuojamų metodikų neturi išsamių nurodymų, kaip turėtų būti užmezgama komunikacija tarp mikroservisų ir pritaikoma duomenų bazė mikroservisų architektūrai. Monolitinės programos kodo kokybė turi didelę įtaką perkėlimo procesui. Kuo geresnė kokybė, tuo mažiau pastangų reikia norint pereiti nuo monolitinės prie mikroservisų architektūros.

Apibendrinant galima teigti, kad mikroservisų architektūra turi daug pranašumų, lyginant su monolitine architektūra, ir daugelyje įmonių tapo standartine architektūra kuriant šiuolaikinę debesų kompiuterija grįstą programinę įrangą. Daugelis įmonių pradėjo modernizuoti savo esamas monolitines programas, išskaidydamos jas į mikroservisus, siekdamos išlaikyti konkurencingumą rinkoje. Pažymėtina, kad mikroservisų architektūra yra sudėtingas, kompleksinis ir palyginti naujas architektūros stilius. Nėra plačiai patvirtinto būdo, kaip atlikti perkėlimą iš monolitinės architektūros į mikroservisų architektūrą. Nustatyti trys pagrindiniai iššūkiai migruojant iš monolitinės į mikroservisų architektūrą: mikroservisų identifikavimas ir išgavimas iš monolitinių programų išeities kodų bazių, ryšio tarp išskaidytų mikroservisų užmezgimas, duomenų bazių pritaikymas mikroservisų architektūrai. Nors mikroservisų identifikavimas ir išgavimas iš išeities kodo yra plačiai tyrinėtas mokslininkų ir programinės įrangos inžinierių, tačiau komunikacijos užmezgimas tarp mikroservisų ir duomenų bazės pritaikymas mikroservisų architektūrai vis dar yra mažai tyrinėtas. Kiekvienas mikroservisas gali būti skirtingas įvairiais aspektais ir nėra vienos duomenų bazės, kuri potencialiai galėtų patenkinti visus poreikius, todėl natūralu, kad daugiamodelinė poliglotinė duomenų bazės technologija tampa puikiu pasirinkimu siekiant išnaudoti mikroservisų architektūros teikiamos privalumus modernizuojant monolitinę duomenų bazę.

2. Perkėlimo iš monolitinės į mikroservisų architektūrą metodas

Šiame skyriuje apžvelgiamas siūlomas perkėlimo metodas, leidžiantis perkelti esamą monolitinę programą į mikroservisų architektūrą. Perkėlimas iš monolitinės architektūros į mikroservisų architektūrą yra sudėtingas kompleksinis iššūkis, sudarytas iš daugybės skirtingų problemų, tokių kaip mikroservisų identifikavimas, išeities kodo išskaidymas, mikroservisų komunikacijos užmezgimas, nepriklausomas diegimas, duomenų saugojimo pritaikymas ir kt. Skirtingai nuo kitų pasiūlytų migracijos metodų, siūlomas metodas susideda iš trijų dalių: išeities kodo išskaidymo į mikroservisus, ryšio užmezgimo tarp išskaidytų mikroservisų ir duomenų bazės adaptacijos prie mikroservisų architektūros. Pažymėtina, jog daugumos kitų tyrimų pagrindinis dėmesys skiriamas mikroservisams identifikuoti monolitinėje programoje ir išeities kodo išskaidymui į mikroservisus. Pabrėžtina, jog esami perkėlimo metodai pateikia labai mažai arba visai nepateikia rekomendacijų, kaip pritaikyti duomenų saugyklą prie mikroservisų architektūros ir kaip užmegzti ryšį tarp mikroservisų iš monolitinės architektūros į mikroservisų architektūrą perkėlimo metu.

Pagrindiniai siūlomo perkėlimo iš monolitinės architektūros į mikroservisų architektūrą metodo žingsniai parodyti S2.1 paveiksle. Šį metodą sudaro penki pagrindiniai žingsniai, kurių kiekvienas yra padalintas į keletą poveiksmių: 1 žingsnis – esamos monolitinės programos analizė; 2 žingsnis – išeities kodo išskaidymas į mikroservisus; 3 žingsnis – ryšio tarp išskaidytų mikroservisų užmezgimas; 4 žingsnis – duomenų bazės pritaikymas mikroservisų architektūrai; 5 žingsnis – išleidimas ir diegimas.


Žingsnis 1.
Esamos monolitinės programos analizė
Žingsnis 2.
Monolitinės programos kodo išskaidymas į mikroservisus
Žingsnis 3.
Komunikacijos užmezgimas tarp mikroservisų
Žingsnis 4.
Duomenų bazės adaptacija
Žingsnis 5.
Diegimas
Požingsnis 1.
Kodo skaidymo reikalavimų nustatymas
Požingsnis 1.
Kodo skaidymo metodo pasirinkimas
Požingsnis 1.
Komunikacijos technologijos pasirinkimas
Požingsnis 1.
Duomenų modelio sukūrimas
Požingsnis 1.
Infrastruktūros sukūrimas
Požingsnis 2.
Komunikacijos reikalavimų nustatymas
Požingsnis 2.
Kodo išskaidymo į mikroservisus įgyvendinimas
Požingsnis 2.
Komunikacijos užmezgimo įgyvendinimas
Požingsnis 2.
Mikroserviso sukūrimas
Požingsnis 2.
Mikroserviso diegimas
Požingsnis 3.
Duomenų valdymo reikalavimų nustatymas
Požingsnis 3.
Duomenų transformacija
Požingsnis 3.
Hiperpriežiūra
Požingsnis 4.
Duomenų validacija
Požingsnis 4.
Monolitinės programos sustabdymas

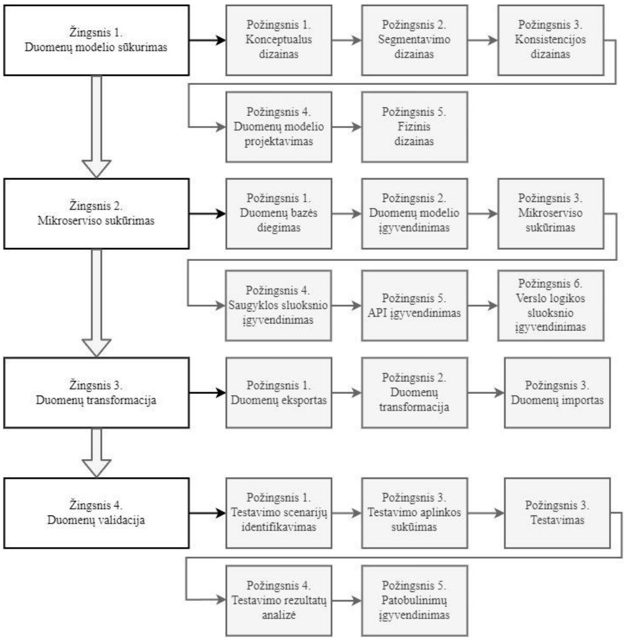
S2.1 pav. Pasiūlytas migracijos iš monolitinės architektūros į mikroservisų architektūrą metodas

Pirmojo žingsnio tikslas yra išanalizuoti esamą monolitinę programą ir identifikuoti funkcinius ir nefunkcinius reikalavimus tolesniems žingsniams. Turi būti surinkti trijų tipų reikalavimai: išeities kodo išskaidymo, ryšio tarp mikroservisų užmezgimo ir duomenų bazės pritaikymo mikroservisų architektūrai.

Antrame žingsnyje reikia parinkti kodo išskaidymo metodą ir juo remiantis išskaidyti esamą monolitinę programą į mikroservisus. Taikant siūlomą metodą numatomi trys išeities kodo išskaidymo metodai, iš kurių galima pasirinkti: *išeities kodo elementais pagrįstas, duomenų bazės elementais pagrįstas* ir *verslo domenais pagrįstas*. Išsamiau apie metodus ir jų vertinimus galima rasti 1 ir 2 disertacijos skyriuose. Pagrindiniai kriterijai renkantis išeities kodo išskaidymo metodą turėtų būti numatyti mikroservisų dydis ir atsakomybių ribos.

Pagrindinis trečiojo žingsnio tikslas yra parinkti komunikacijos technologiją ir užmegzti ryšį tarp mikroservisų, išskaidytų iš monolitinės programos antrajame žingsnyje. Siūlomas metodas leidžia pasirinkti iš penkių komunikacijos technologijų: HTTP Rest, RabbitMQ, Kafka, gRPC ir GraphQL. Taikant siūlomą metodą numatomi kriterijai, kuriais remiantis turėtų būti parinkta komunikacijos technologija. Jei pagrindiniai kriterijai yra delsa ir pralaidumas, tai RabbitMQ ir gRPC yra tinkamiausios technologijos. RabbitMQ labiausia tinkama RPC žinutėms iki 0.1 MB, gRPC labiausiai tinka RPC žinutėms, turinčioms daugiau nei 10000 laukų. Kafka parodė geriausius pralaidumo rezultatus labiausiai apkrautomis sąlygomis. Jei pranešimo dydis yra svarbus kriterijus renkantis komunikacijos technologiją, tuomet HTTP Rest yra rekomenduojama technologija. Jei naudojamos atminties dydis yra vienas iš esminių kriterijų, tada komunikacijai tarp mikroservisų turi būti naudojamos RabbitMQ arba Kafka technologijos.

Ketvirtajame žingsnyje esama monolitinė duomenų bazė turi būti pritaikyta mikroservisų architektūrai. Pasiūlytas metodas leidžia transformuoti ir perkelti monolitinę duomenų bazę į daugiamodelinę poliglotinę duomenų bazę. Transformuota duomenų bazė yra inkapsuliuojama atskirame mikroservise ir priėjimas prie duomenų kitiems mikroservisams yra leidžiamas tik per API. Siūlomas duomenų bazės migracijos metodas yra pateiktas S2.2 paveiksle.


Žingsnis 1. Duomenų modelio sūkurimas
Požingsnis 1. Konceptualus dizainas
Požingsnis 2. Segmentavimo dizainas
Požingsnis 3. Konsistencijos dizainas
Požingsnis 4. Duomenų modelio projektavimas
Požingsnis 5. Fizinis dizainas
Žingsnis 2. Mikroserviso sukūrimas
Požingsnis 1. Duomenų bazės diegimas
Požingsnis 2. Duomenų modelio įgyvendinimas
Požingsnis 3. Mikroserviso sukūrimas
Požingsnis 4. Saugyklos sluoksnio įgyvendinimas
Požingsnis 5. API įgyvendinimas
Požingsnis 6. Verslo logikos sluoksnio įgyvendinimas
Žingsnis 3. Duomenų transformacija
Požingsnis 1. Duomenų eksportas
Požingsnis 2. Duomenų transformacija
Požingsnis 3. Duomenų importas
Žingsnis 4. Duomenų validacija
Požingsnis 1. Testavimo scenarijų identifikavimas
Požingsnis 3. Testavimo aplinkos sukūimas
Požingsnis 3. Testavimas
Požingsnis 4. Testavimo rezultatų analizė
Požingsnis 5. Patobulinimų įgyvendinimas

S2.2 pav. Siūlomas duomenų bazės migracijos metodas

Pirmajame duomenų bazės migracijos metodo žingsnyje, remiantis apibrėžtu esamos monolitinės duomenų bazės modeliu, turi būti sukurtas daugiamodelinio poliglotinio patvarumo duomenų modelis. Pagrindiniai antrojo žingsnio tikslai yra sukurti daugiamodelinę poliglotinę duomenų bazę ir inkapsuliuoti ją į atskirą mikroservisą. Tai leidžia įdiegti duomenų bazę kaip paslaugų modelį, kai duomenų bazė pati yra mikroservisas. Trečiajame žingsnyje duomenys iš esamos monolitinės duomenų bazės turi būti transformuoti ir perkelti į daugiamodelinę poliglotinę duomenų bazę. Paskutinio žingsnio tikslas yra transformuotų duomenų validacija.

Paskutinio žingsnio tikslas – išleisti ir įdiegti sukurtus mikroservisus ir daugiamodelinę poliglotinę duomenų bazę. Tai apima visus techninius ypatumus, reikalingus mikroservisams ir duomenų bazei įdiegti ir paleisti.

3. Mikroservisų komunikacijos tyrimas

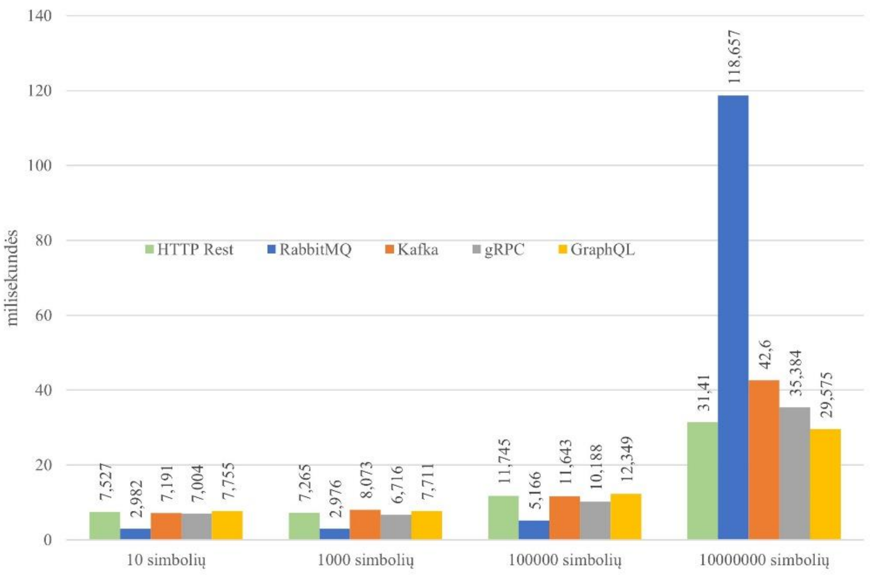
Vienas didžiausių iššūkių pereinant nuo monolitinės architektūros prie mikroservisų architektūros yra pasirinkti tinkamą komunikacijos technologiją. Monolitinėse programose komunikacija tarp komponentų vykdoma naudojant funkcijų iškvietimus. Mikroservisų architektūra pagrįsta programa yra paskirstyta sistema, veikianti keliuose procesuose, todėl mikroservisai turi sąveikauti naudodami tarp procesines komunikacijos technologijas. Šiame skyriuje įvertinamos skirtingos komunikacijos technologijos ir nustatomi konkretūs jų taikymo atvejai, išskaidant monolitą į mikroservisus. Penkios komunikacijos technologijos, tokios kaip HTTP Rest, RabbitMQ, Kafka, gRPC ir GraphQL, buvo įvertintos ir palygintos pagal siūlomus vertinimo kriterijus: greitaveika (delsa ir pralaidumas), žinutės dydis, naudojamas operatyviosios atminties kiekis, naudojamas saugyklos atminties kiekis, paleidimo laikas, architektūra, topologija ir naudojamos bibliotekos.

Komunikacijos technologijoms įvertinti ir palyginti buvo sukurtas ir linijine topologija sujungtas penkių mikroservisų rinkinys. Ryšiui tarp mikroservisų buvo naudojama RPC technika. Eksperimentu buvo siekiama įvertinti ir palyginti ryšį, pagrįstą nuotoliniu procedūrų iškvietimu (RPC). RPC technika buvo pasirinkta, nes ji palaiko tą patį funkcionalumą kaip funkcijos iškvietimas. Skirtingos žinutės buvo naudojamos eksperimento metu nustatyti žinutės dydžio ir kompleksiškumo įtaką delsos ir pralaidumo parametrams. Visi mikroservisai buvo parašytos naudojant C# programavimo kalbą. Kodo rašymas ir testavimas buvo atliktas naudojant „Microsoft Visual Studio 2022 IDE“. Delsos testai buvo atlikti naudojant „BenchmarkDotNet“ biblioteką. Pralaidumo testai buvo atlikti naudojant „NBomber“ biblioteką. Tinklo duomenys buvo išanalizuoti „Wireshark“ programa.

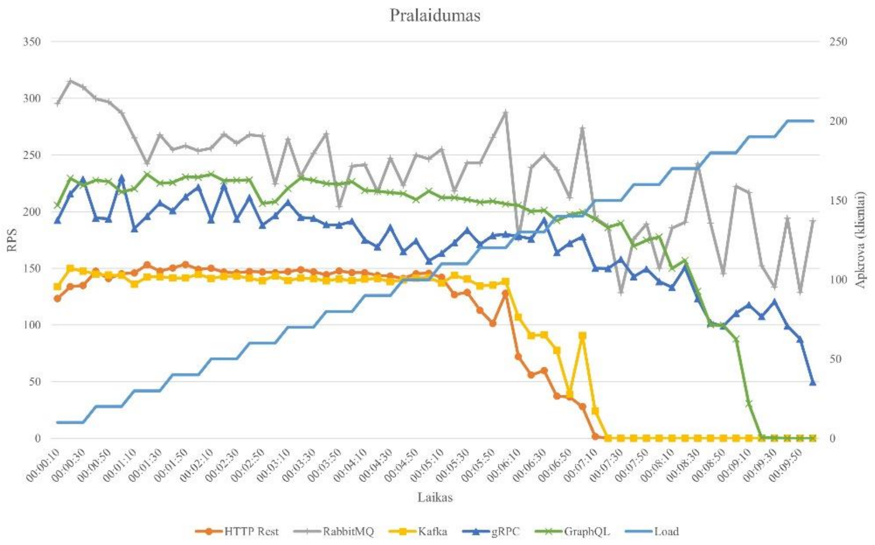
Geriausi delsos rezultatai žinutėms iki 1 000 000 simbolių buvo gauti naudojant RabbitMQ technologiją (S3.1 pav.). RabbitMQ buvo 2 kartus greitesnis nei kitos technologijos. Jis parodė geriausius rezultatus apdorojant mažiausius pranešimus (10 ir 1000 simbolių). HTTP Rest, Kafka, gRPC ir GraphQL rodė panašius delsos rezultatus, tačiau gRPC gauti rezultatai buvo šiek tiek geresni. Kita vertus, RabbitMQ turėjo blogiausius delsos rezultatus apdorojant pranešimus, kuriuos sudarė 10 000 000 simbolių. Jis buvo nuo 3 iki 4 kartų lėtesnis nei kiti. Geriausi 10 000 000 simbolių pranešimų delsos rezultatai buvo gauti naudojant GraphQL ir HTTP Rest technologijas. Kafka buvo 40 %, o gRPC – 16 % ir lėtesnis nei GraphQL ir HTTP Rest technologijos.

Mažiausio kompleksiškumo žinutėms, kuriose buvo iki 1000 laukų, geriausi delsos rezultatai taip pat buvo gauti naudojant RabbitMQ technologiją. RabbitMQ delsos rezultatai buvo nuo 2 iki 3 kartų greitesni nei kitos technologijos. Geriausi rezultatai bendraujant žinutėmis, kuriose buvo 10 000 laukų, buvo gauti naudojant gRPC technologiją. GRPC technologijos naudojamas dvejetainis serializavimas yra greitesnis nei JSON serializavimas, kurį eksperimento metu naudojo kitos technologijos, todėl kuo daugiau laukų pranešimas turi, tuo didesnis gRPC pranašumas.

10 simbolių dydžio pranešimo pralaidumo rezultatai parodyti S3.2 paveiksle. Geriausi našumo rezultatai buvo gauti naudojant RabbitMQ technologiją, vidutinis 231,6 RPS. Maksimalus rezultatas – 315,1 RPS – pasiektas kreipiantis su 10 klientų. Blogiausius RPC pralaidumo testo rezultatus gavo HTTP Rest technologija su vidutiniu 89,8 RPS ir 140 klientų limitu.



S3.1 pav. Delsos testų rezultatai



S3.2 pav. Pralaidumo testų rezultatai 10 simbolių žinutėms

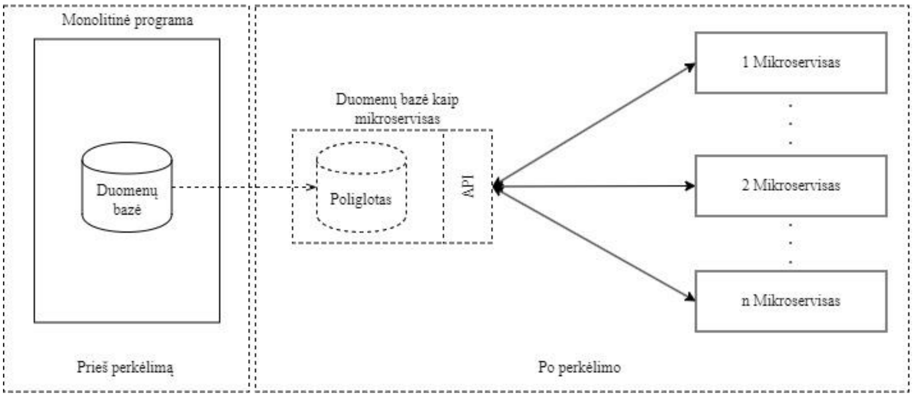
Galima apibendrinti, kad geriausi RPC pralaidumo rezultatai mažesniems pranešimams, iki 0,1 MB ir iki 100 laukų, buvo pasiekti naudojant RabbitMQ technologiją. Geriausi RPC pralaidumo rezultatai didesniems pranešimams buvo pasiekti naudojant gRPC ryšio technologiją. Naudojant Kafka technologiją buvo pasiekti prasčiausi pralaidumo rezultatai – 5 iš 8 atvejų. Lėčiausia technologija, apdorojanti didžiausias žinutes, 1 000 000 simbolių, buvo RabbitMQ.

Tačiau palyginus delsos pasiskirstymo rezultatus matyti, kad tiek Kafka, tiek RabbitMQ technologijos gali apdoroti daugiau pranešimų (su delsa, didesne nei 1 sekundė) ir veikia stabiliau bendradarbiaudamos su daugiau nei 50 klientų, palyginti su HTTP Rest, gRPC ir GraphQL technologijomis.

4. Monolitinės duomenų bazės perkėlimo į daugiamodelinę poliglotinę duomenų bazę tyrimas

Perkėlimas iš monolitinės architektūros į mikroservisų architektūrą yra sudėtingas ir kompleksinis procesas. Vienas iš pagrindinių iššūkių yra duomenų bazės pritaikymas prie mikroservisų architektūros. Monolitinėje architektūroje programa sąveikauja su viena duomenų baze, o mikroservisų architektūroje duomenų saugojimas yra decentralizuotas – kiekvienas mikroservisas veikia savarankiškai. Poliglotinė duomenų bazių technologija puikiai tinka mikroservisų architektūrai patenkinti skirtingas, skirtingų mikroservisų duomenų saugojimo poreikio, ypatybes.

Šiame skyriuje įvertinamas siūlomas monolitinės duomenų bazės perkėlimo į daugiamodelinę poliglotinę duomenų bazę, pritaikytą mikroservisų architektūrai, metodas. Perkėlimas iš egzistuojančios monolitinės duomenų bazės į daugiamodelinę poliglotinę duomenų bazę buvo atliktas kaip siūlomo migracijos metodo koncepcijos įrodymas. Kokybės atributai, apibrėžti standarte ISO/IEC 25012:2008, buvo naudojami vertinant ir lyginant mikroservisų architektūra grįstą daugiamodelinę poliglotinę ir esamą monolitinę duomenų bazes. Tyrimo rezultatai parodė, kad siūlomas metodas gali būti naudojamas atliekant duomenų bazės perkėlimą iš monolitinės architektūros į mikroservisų architektūrą bei pagerinti nuoseklumo, suprantamumo, prieinamumo ir perkeliamumo atributų kokybę.



S4.1 pav. Siūlomo duomenų bazių perkėlimo metodo tikslas

Siūlomo metodo tikslas pateiktas S4.1 paveiksle. Siūlomas metodas suteikia galimybę transformuoti ir perkelti monolitinę duomenų bazę į daugiamodelinę poliglotinę duomenų bazę. Transformuota duomenų bazė yra inkapsuliuojama atskirame mikroservise ir priėjimas prie duomenų kitiems mikroservisams yra leidžiamas tik per taikomųjų programų programavimo sąsają. Daugiamodelinė poliglotinė duomenų bazė leidžia geriau išnaudoti mikroservisų architektūros pranašumus, tokius kaip judrumas ir mastelio keitimas. Duomenų bazės įtraukimas į mikroservisą sumažina sudėtingumą ir padidina našumą. Atlikus duomenų perkėlimą, jie tampa pasiekiami ne tik esamai monolitinei programai, bet ir bet kuriam ekosistemos mikroservisui. Sukurtas duomenų pasiekiamumas, suteikia galimybę palaipsniui dekomponuoti išeities kodą iš monolitinės į mikroservisų architektūrą.

Duomenų kokybė yra pagrindinis kriterijus, nusakantis informacinių sistemų kokybę ir naudingumą. Verslo procesų efektyvumas tiesiogiai priklauso nuo duomenų kokybės. Šiame skyriuje pateikiami ISO/IEC 25012:2008 standarto kokybės atributų įvertinimo ir palyginimo rezultatai tarp esamos monolitinės duomenų bazės ir mikroservisų architektūra pagrįstos daugiamodelinės poliglotinės duomenų bazės. Kiekvienas kokybės požymis buvo įvertintas balais nuo 1 iki 5. Mažesnė balo vertė rodo žemesnę kokybę, o didesnė balo vertė – aukštesnę kokybę. Naudotų vertinimo balų aprašymai pateikti S4.1 lentelėje.

S4.1 lentelė. Balai, naudoti kokybės požymiams įvertinti

Balo vertė	**Apibrėžimas**
1	Žemiausia kokybė
2	Žema kokybė
3	Vidutiniška kokybė
4	Aukšta kokybė
5	Aukščiausia kokybė

Vertinimas buvo atliktas organizacijos verslo srities ekspertų ir IT ekspertų forume. Forume dalyvavo trys domeno ekspertai, keturi monolitinės programinės įrangos inžinieriai ir keturi C# programinės įrangos inžinieriai. Buvo pateikta 150 klausimų, po 10 klausimų kiekvienam kokybės požymiui. Kiekvienas klausimas buvo taikomas abiem duomenų bazėms. Fleiso Kapos κ koeficientas buvo naudojamas ekspertų susitarimui įvertinti (Fleiss et al., 2003). Koeficiento reikšmė buvo 0,77, o tai rodo gana aukštą ekspertų sutarimo lygį. Galutiniai vertinimo ir palyginimo rezultatai pateikti S4.2 lentelėje. Galutinė kiekvienos kokybės atributo vertė yra įverčių vidurkis, suapvalintas iki artimiausio sveikojo skaičiaus.

S4.2 lentelė. ISO/IEC 25012:2008 standarto kokybės atributų tarp monolitinės ir mikroservisų duomenų bazių įvertinimo ir palyginimo rezultatai

Kokybės atributas	**Monolitas**	**Mikroservisas**
Tikslumas	5	5
Išsamumas	5	5
Nuoseklumas	3	5
Patikimumas	5	5

S4.2 lentelės pabaiga

Kokybės atributas	**Monolitas**	**Mikroservisas**
Teisingumas	4	4
Prieinamumas	4	4
Atitikimas	5	5
Konfidencialumas	5	5
Efektyvumas	4	4
Tikslumas	5	5
Atsekamumas	5	5
Supratimas	3	5
Prieinamumas	2	4
Perkeliamumas	1	5
Atkuriamumas	4	4

Dauguma ISO/IEC 25012:2008 standarto kokybės atributų, tokių kaip tikslumas, išsamumas, patikimumas, teisingumas, prieinamumas, atitiktis, konfidencialumas, efektyvumas, tikslumas, atsekamumas ir atkuriamumas, buvo vienodi abiem duomenų bazėms, tačiau mikroservisų architektūra pagrįstos daugiamodelinės poliglotinės duomenų bazės parodė geresnius nuoseklumo, suprantamumo, prieinamumo ir perkeliamumo rezultatus.

Bendrosios išvados

1. Atlikta literatūros apžvalga parodė, kad mikroservisų architektūra tampa *de facto* pramonės standartu kuriant naujas programas. Siekdamos išlikti konkurencingos, įmonės pradėjo modernizuoti savo senas monolitines sistemas, išskaidydamos jas į mikroservisus. Tačiau perėjimas nuo monolitinės architektūros prie mikroservisų architektūros yra sudėtingas iššūkis, kurį sudaro tokios problemos kaip mikroservisų identifikavimas, kodo išskaidymas, nepriklausomas diegimas ir kt. Kiekviena įmonės programa yra unikali. Ji buvo programuojama naudojant skirtingas programavimo kalbas ir technologijas, buvo naudojamos skirtingos duomenų bazės ir komunikacijos mechanizmai, todėl tai kelia skirtingus iššūkius. Nors monolitinės programinės įrangos perkėlimo į mikroservisų architektūrą aktualijas jau tyrinėjo mokslininkai ir programinės įrangos inžinieriai, tai sudėtingas ir palyginti naujas iššūkis, todėl daugelis jo dalių vis dar mažai tyrinėjamos, pavyzdžiui: duomenų bazės pritaikymas ir komunikacijos tarp mikroservisų užmezgimas. Pagrindinis daugumos tyrimų dėmesys skiriamas mikroservisams identifikuoti monolitinėje programoje ir šaltinio kodui išskaidyti į mikroservisus.
2. Siekiant užpildyti spragas komunikacijos ir duomenų bazių srityse buvo pasiūlytas naujas perkėlimo metodas, pagrįstas eksperimentiniais tyrimais. Šis metodas apima tris pagrindinius elementus: kodo išskaidymo būdus, ryšio sukūrimą ir duomenų bazės pritaikymą. Inovatyvūs vertinimo kriterijai ir gairės, paimtos iš empirinių išvadų, padeda rekomenduoti tinkamiausią kodų skaidymo metodą ir komunikacijos technologiją, atsižvelgiant į jų privalumus ir trūkumus. Siekiant

palengvinti duomenų bazės perėjimą prie mikroservisų architektūros, buvo pasiūlytas naujas duomenų bazės perkėlimo metodas, kurį taikant naudojamas kelių modelių poliglotinis duomenų saugojimo modelis, ir įvertintas atliekant eksperimentinį vertinimą.

3. Išanalizuoti ir palyginti pasirinkti trys monolitinės architektūros programos išeities kodų skaidymo į mikroservisus metodai: *Duomenų bazės elementais pagrįstas metodas, Kodo elementais pagrįstas metodas*, *Verslo domenu pagrįstas metodas*. Pasirinktų metodų palyginimas buvo atliktas tris kartus išskaidžius tą pačią monolitinę programą į mikroservisus, taikant visus pasirinktus metodus.
   3.1. Kodo ir duomenų bazių elementais pagrįsti mikroservisų identifikavimo metodai leidžia identifikuoti monolitinės programos technines funkcijas ir priskirti joms atitinkamus kodo ir duomenų bazių komponentus, o verslo domenais pagrįsti mikroservisų identifikavimo metodai leidžia identifikuoti mikroservisus pagal identifikuotas verslo sritis. Mikroservisai, pagrįsti techninėmis funkcijomis, užtikrina didesnį detalumą.
   3.2. Verslo domenu grįstas metodas ir kodo elementais grįstas metodas su semantinio susiejimo strategija turėtų būti taikomas monolitinei programai išskaidyti į mikroservisus, paremtus atskirais verslo domenais.
   3.3. Duomenų bazės elementais grįstas metodas arba kodo elementais grįstas metodas su loginio susiejimo strategija turėtų būti taikomas, norint išskaidyti monolitinę programą į mikroservisus, paremtus techninėmis funkcijomis.
4. Penkios komunikacijos technologijos, HTTP Rest, RabbitMQ, Kafka, gRPC ir GraphQL, buvo įvertintos ir palygintos pagal siūlomus vertinimo kriterijus. Kiekvienos komunikacijos technologijos privalumai ir trūkumai buvo nustatyti mikroservisų architektūros kontekste.
   4.1. Pereinant nuo monolitinės architektūros prie mikroservisų architektūros pagrindiniai kriterijai yra delsa ir pralaidumas, o RabbitMQ ir gRPC yra tinkamiausios technologijos. RabbitMQ parodė geriausius delsos ir pralaidumo testų rezultatus žinutėms iki 0,1 MB, o gRPC parodė geriausius rezultatus bendraujant žinutėmis, turinčiomis daugiau kaip 1000 laukų.
   4.2. Kafka ir RabbitMQ parodė geriausius pralaidumo rezultatus labiausiai apkrautomis sąlygomis, tačiau delsos laikas buvo didesnis nei 1 sekundė.
   4.3. HTTP Rest turi mažiausią užklausos ir atsakymo pranešimo dydį. Jei pranešimo dydis yra svarbus kriterijus renkantis komunikacijos technologiją, tada HTTP Rest yra rekomenduojama technologija.
   4.4. gRPC biblioteka naudoja mažiausiai saugyklos vietos. Jei mikroservisai veikia aplinkoje su ribota saugykla, reikia naudoti gRPC.
   4.5. RabbitMQ ir Kafka naudoja mažiausią operatyviosios atminties kiekį. Todėl, jei operatyviosios atminties dydis yra vienas iš esminių kriterijų, diegimui reikia naudoti RabbitMQ ir Kafka.
5. Pasiūlytas monolitinės duomenų bazės perkėlimas į daugiamodelinę poliglotinę duomenų bazę, paremtas mikroservisų architektūra, atliktas kaip koncepcijos įrodymas ir įvertintas domenų ir IT ekspertų. Ekspertų sutarimui įvertinti buvo

naudojamas sutarimas tarp vertintojų Fleiss kappa κ (Fleiss et al., 2003). Koeficiento reikšmė buvo 0,77, o tai rodo gana aukštą ekspertų sutarimo lygį. Tyrimo rezultatai parodė, kad siūlomas metodas gali būti taikomas duomenų saugyklai perkelti iš monolitinės į mikroservisų architektūrą ir nuoseklumo, nesupratimo, prieinamumo ir perkeliamumo atributų kokybei pagerinti. Be to, tikimasi, kad gauti rezultatai galėtų įkvėpti tyrėjus ir praktikus tolesniam darbui, siekiant pagerinti ir automatizuoti siūlomą metodą.

Annexes

The questionnaire of the evaluation of the data quality of the proposed microservice with multi–model polyglot persistence is provided in Tables A1.1–A1.15.

Table A1.1. Accuracy attribute questions

Nr.	Question
1	Are the names and details of the items in the database correct and up-to-date?
2	Does the database provide accurate information when you search for something?
3	Are the numbers and calculations in the database correct, without errors or miscalculations?
4	Are dates and times in the database accurate, reflecting the real-world events they represent?
5	Do you trust the data in the database to make informed decisions?
6	Have you encountered any instances where the information in the database contradicts real-world facts?
7	Are there mechanisms in place to prevent or correct errors in the database?
8	Can you rely on the database to give you a clear picture of what is happening in a specific situation?
9	Have you noticed any inconsistencies or discrepancies between different parts of the database?
10	Is there a process for regularly checking and ensuring the accuracy of the data in the database?

Table A1.2. Completeness attribute questions

Nr.	Question
1	Does the database contain all the necessary information you expect to find?
2	Are there any gaps or missing details in the data that you need?
3	Are there placeholders or placeholders for missing information in the database?
4	Are dates and times in the database accurate, reflecting the real-world events they represent?
5	Has anyone encountered situations where they couldn't find the data they were looking for?
6	Is the database regularly updated to include new and relevant information?
7	Are there any areas in the database where information seems to be lacking or incomplete?
8	Can you trust that the data in the database gives you a full picture of a particular situation?
9	Have you experienced instances where the database lacks details about specific events or items?
10	Is there a process in place to identify and fill in missing information in the database?

Table A1.3. Consistency attribute questions

Nr.	Question
1	Do you notice any conflicting information or contradictions within the database?
2	Are there instances where terms or units vary inconsistently throughout the database?
3	Does the database maintain a standardized and consistent format for presenting information?
4	Have you encountered situations where the same data appears differently in different sections of the database?
5	Is there a clear and consistent approach to handling data across various parts of the database?
6	Are there established rules for data entry and storage to ensure overall consistency?
7	Does the database use consistent terminology and definitions for similar data elements?
8	Have you observed any discrepancies in how dates and times are formatted or recorded?
9	Is there a process in place to resolve inconsistencies and ensure data uniformity?
10	Are users provided with guidelines to maintain consistency when entering or updating data in the database?

Table A1.4. Credibility attribute questions

Nr.	Question
1	Can you trust the accuracy of the information stored in the database?
2	Have you encountered situations where the database provided misleading or inaccurate data?
3	Is there a clear source or origin documented for the information in the database?
4	Are there measures in place to verify and validate the data before it is entered into the database?
5	Does the database provide information about the reliability of its sources?
6	Are there mechanisms to identify and flag potentially unreliable or outdated information?
7	Have users experienced instances where they questioned the trustworthiness of the database data?
8	Is there a process to regularly review and update information to maintain credibility?
9	Does the database follow industry standards for data quality and credibility?
10	Are there user permissions or access controls to prevent unauthorized modifications that could impact credibility?

Table A1.5. Correctness attribute questions

Nr.	Question
1	Are the names and details of items in the database accurate and error-free?
2	Do calculations and numerical data in the database appear correct without miscalculations?
3	Are dates and times accurately represented in the database, reflecting real-world events?
4	Has the database been reliable in providing accurate information when searched or queried?
5	Is there a process to verify and validate data before it is entered into the database?
6	Have users experienced any situations where the database contained incorrect or misleading information?
7	Are there mechanisms in place to identify and correct errors or discrepancies in the database?
8	Can you trust the data in the database to make informed decisions without concerns about correctness?
9	Is there a standardized approach to data entry and storage to ensure correctness?
10	Are there regular audits or checks to ensure the overall correctness of the information in the database?

Table A1.6. Accessibility attribute questions

Nr.	Question
1	Can authorized users easily access the database when needed?
2	Is the interface of the database user-friendly for individuals with varying technical backgrounds?
3	Are there restrictions or barriers preventing certain users from accessing specific data?
4	Can the database be accessed from different devices or locations without difficulty?
5	Is there a support system in place to assist users with accessing and navigating the database?
6	Are there clear guidelines on how to request access or permissions for specific database features?
7	Does the database provide options for accessibility features, such as screen readers or keyboard navigation?
8	Have users experienced any challenges in accessing specific functionalities within the database?
9	Is there a process for securely sharing or distributing relevant information from the database to authorized users?
10	Are there measures in place to protect sensitive data and ensure secure access to the database?

Table A1.7. Compliance attribute questions

Nr.	Question
1	Does the database adhere to relevant legal regulations and industry standards?
2	Are there documented policies outlining the compliance requirements for the database?
3	Has the database undergone audits or assessments to ensure compliance with standards?
4	Are there mechanisms in place to monitor and address changes in compliance regulations?
5	Does the database provide clear documentation on data handling and privacy practices?
6	Are there measures to ensure that the database complies with data protection laws?
7	Is user access to sensitive information controlled to meet privacy and security standards?
8	Does the database have features to support compliance reporting and documentation?
9	Are there procedures in place to address and rectify any non-compliance issues promptly?
10	Has the database been designed and maintained with considerations for ethical and legal data usage?

Table A1.8. Confidentiality attribute questions

Nr.	Question
1	Are there measures in place to safeguard sensitive information from unauthorized access?
2	Does the database use encryption to protect confidential data during storage and transmission?
3	Are there access controls to restrict user access based on their roles and responsibilities?
4	Is there a clear policy outlining the handling of confidential information within the database?
5	Are user authentication mechanisms in place to ensure that only authorized users can access sensitive data?
6	Has the database undergone security assessments to identify and address potential vulnerabilities?
7	Are there procedures for securely sharing confidential information with authorized parties?
8	Is there a system for monitoring and detecting any unauthorized attempts to access confidential data?
9	Have there been incidents of data breaches or unauthorized access to confidential information?
10	Is there ongoing training for users on the importance of maintaining the confidentiality of data in the database?

Table A1.9. Efficiency attribute questions

Nr.	Question
1	Does the database efficiently handle a large volume of data without significant performance degradation?
2	Are there features or tools to optimize and improve the overall performance of the database?
3	Have users experienced delays or slowdowns when interacting with the database?
4	Is there a process for periodically tuning the database to maintain optimal performance?
5	Does the database efficiently manage and allocate system resources to avoid bottlenecks?
6	Are there measures in place to identify and address performance issues promptly?
7	Has the database been designed with considerations for scalability to accommodate future growth?
8	Is there documentation available on best practices for maximizing the efficiency of the database?
9	Does the database efficiently handle a large volume of data without significant performance degradation?
10	Are there features or tools to optimize and improve the overall performance of the database?

Table A1.10. Precision attribute questions

Nr.	Question
1	Does the database provide accurate and detailed information with a high level of precision?
2	Are there clear definitions and standards for the precision of numerical values in the database?
3	Does the database avoid rounding errors or inaccuracies in calculations involving numerical data?
4	Are there measures to ensure that data with a specific level of precision is consistently maintained?
5	Have users encountered situations where the precision of data was insufficient for their needs?
6	Is there a documented policy or guideline on maintaining precision in the database?
7	Are there tools or features in place to support precise data entry and validation?
8	Does the database handle decimal points and significant figures accurately?
9	Is there a process for reviewing and correcting precision-related issues in the database?
10	Have there been instances where the precision of data impacted decision-making or analysis?

Table A1.11. Traceability attribute questions

Nr.	Question
1	Is there a clear trail or record of changes made to the data in the database?
2	Can you trace the origin or source of specific information stored in the database?
3	Does the database provide an audit trail for data modifications and updates?
4	Are there mechanisms to track and trace the flow of data through different processes in the database?
5	Is there documentation on how data is transformed and transferred within the database?
6	Can users easily identify the relationships and dependencies between different data elements?
7	Does the database maintain a history of changes, allowing for rollback or recovery if needed?
8	Are there tools or features in place to support effective data lineage and traceability?
9	Is there a process for documenting and managing the relationships between different data sets?
10	Have users experienced difficulties in tracing the history or lineage of specific data elements?

Table A1.12. Understandability attribute questions

Nr.	Question
1	Is the data model easily understandable?
2	Can users easily comprehend the meaning and purpose of different data elements in the database?
3	Are there clear and concise labels used for fields and categories in the database?
4	Does the database provide documentation or guides to help users understand its structure and use?
5	Have users encountered difficulties in interpreting or navigating the database?
6	Is there a standardized format for presenting information that enhances user comprehension?
7	Does the database use terminology that is familiar and easily understood by its users?
8	Are there tooltips or contextual help features to assist users in understanding specific elements?
9	Is there a process for user feedback and improvement based on user understanding challenges?
10	Have there been instances where misunderstandings of data in the database led to errors or confusion?

Table A1.13. Availability attribute questions

Nr.	Question
1	Has the database been consistently available and accessible when needed?
2	Are there measures in place to prevent or minimize downtime for routine maintenance?
3	Is there a backup and recovery system to ensure data availability in case of unexpected issues?
4	Have users experienced any difficulties accessing the database due to technical issues?
5	Does the database have failover mechanisms to ensure continuous access in case of server failures?
6	Is there a process for monitoring and addressing performance issues that could impact availability?
7	Are there redundant systems or servers to provide backup in case of hardware failures?
8	Is there a documented service level agreement (SLA) outlining expected availability standards?
9	Have there been instances where users were unable to access critical information due to database unavailability?
10	Are there alerts or notifications in place to inform users of planned downtime or maintenance?

Table A1.14. Portability attribute questions

Nr.	Question
1	Can the database be easily migrated or transferred to different platforms or environments?
2	Are there documented procedures for moving the database to a new system or location?
3	Does the database support standard data formats that facilitate interoperability with other systems?
4	Is there compatibility with various operating systems for hosting the database?
5	Can users access and use the database from different devices and locations without major issues?
6	Are there measures in place to handle data migrations seamlessly when upgrading the database?
7	Does the database support standard communication protocols for data exchange?
8	Is there a process for ensuring that third-party applications can integrate smoothly with the database?
9	Have users experienced challenges when attempting to use the database on different platforms?
10	Is there documentation available on best practices for maintaining portability in the database?

Table A1.15. Recoverability attribute questions

Nr.	Question
1	Is there a robust backup and recovery system in place for the database?
2	Can the database be restored to a consistent state after unexpected failures or outages?
3	Are there regular backup procedures to ensure data can be recovered from different points in time?
4	Does the database provide options for partial or full recovery in case of data corruption?
5	Are there mechanisms to detect and repair errors in the database to facilitate recovery?
6	Is there documentation on recovery procedures in case of data loss or system failures?
7	Have users experienced instances where data could not be successfully recovered from backups?
8	Is there a process for testing and validating the effectiveness of the recovery mechanisms?
9	Does the database provide options for disaster recovery to handle major incidents?
10	Are there measures in place to minimize downtime and data loss during the recovery process?

Justas KAZANAVIČIUS

RESEARCH ON LEGACY MONOLITH APPLICATIONS DECOMPOSITION INTO MICROSERVICE ARCHITECTURE

Doctoral Dissertation

Technological Sciences,
Informatics Engineering (T 007)

MONOLITINĖS ARCHITEKTŪROS PROGRAMŲ MIGRACIJOS Į MIKROSERVISŲ ARCHITEKTŪRĄ TYRIMAS

Daktaro disertacija

Technologijos mokslai,
Informatikos inžinerija (T 007)

Lietuvių kalbos redaktorė Dalia Markevičiūtė
Anglų kalbos redaktorė Jūratė Griškėnaitė

2024 04 19. 14,2 sp. l. Tiražas 20 egz.
Leidinio el. versija *https://doi.org/10.20334/2024-018-M*
Vilniaus Gedimino technikos universitetas
Saulėtekio al. 11, 10223 Vilnius
Spausdino UAB „Ciklonas“,
Žirmūnų g. 68, 09124 Vilnius