2022 International Conference on Computing, Electronics & Communications Engineering (iCCECE) | 978-1-6654-8980-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/iCCECE55162.2022.9875096

Architect: A Framework for the Migration to Microservices

Evgeny Volynsky
Technical University of Munich
Munich, Germany
evgeny.volynsky@tum.de

Merlin Mehmed
Technical University of Munich
Munich, Germany
merlin.mehmed@tum.de

Stephan Krusche
Technical University of Munich
Munich, Germany
krusche@in.tum.de

***Abstract*— The migration from a monolithic to a microservice architecture is a recurring step in many software projects. With the increasing distributed nature of the transformed system, new challenges for data consistency and deployment arise, which can be counteracted by the integration of microservices patterns. However, the use of such patterns is complex and time-consuming. In this paper, we describe a case study of a migration process of the learning management system Artemis which consists of two phases. The first phase shows the transformation from a monolithic architecture to a microservice architecture with a shared database. It sets as a goal the identification of microservice boundaries, the decomposition of the monolith application into multiple distributed entities, as well as their orchestration in a cloud-ready environment. The second phase migrates the shared database into multiple databases based on the database-per-microservice pattern. While analyzing the current Artemis architecture, we describe a gradual refactoring of an existing application to decompose Artemis into multiple subsystems. We developed Architect, a framework which is based on a domain-specific language for building dependable distributed systems as a template to ensure the data consistency of the distributed transactions using the Saga pattern. We decomposed Artemis into 3 microservices and provided the migration concept from shared-database to database-per-microservice using Architect. The framework helped to reduce the complexity of using the Saga pattern. It introduced the eventual consistency in a distributed database system and decreased the coupling of the data storage. The migration to a microservice architecture solves many problems of a monolith application, but introduces new challenges and increases the complexity of the overall system. Architect focuses on greenfield project, but currently does not provide a software evolution approach. We will add support for reengineering projects, which can facilitate the migration process of existing system.**

***Index Terms*—architect, microservices, migration, monolith, sagas**

I. Introduction

The monolithic architecture is simple, easy to develop, test and deploy. As the code base grows, it becomes highly coupled, hard to maintain and scale [1]. On the other side, the microservice architecture solves those issues by proposing several small applications that work as a single one. Each microservice is easier to maintain and scale. We deploy the microservices independently, which means we can scale only those microservices that receive more traffic [2]. Moreover, if one microservice fails, the system will continue to operate, and the users can use the microservices which have not failed. Although the microservice architecture solves many of the monolith's problems, it also introduces new challenges for the development and deployment of the system. It becomes more complex to test, debug and deploy. The developers need more knowledge to maintain and develop it [2].

Despite all challenges that the microservice architecture introduces, many companies decide to migrate to the microservice architecture. Their main reason is better system availability and efficient resource usage of a microservice system [3]. They also take advantage of the microservice architecture to allocate small development teams on separate microservice(s) and split the responsibility. This approach helps to reduce the communication overhead and the need for coordination between the teams [4].

II. Motivation

There are two major common reasons why companies decide to migrate to microservices. The first is that the monolith application is hard to maintain, the system takes a long time to start, and it takes more development efforts to make changes due to high coupling and complexity [3]. Another reason is the need for high availability and better scalability to handle the changing traffic dynamically [3].

The case study of this article is the migration to microservices of Artemis – an open-source system for interactive learning. It provides automated assessment tools which help to decrease the tutors' efforts [5]. The initial Artemis architecture is monolithic. Artemis has similar problems as the common ones that we already described. Due to the high user load, Artemis requires better scaling, which the Artemis team has currently solved by deploying several Artemis instances. This scaling approach is not optimal, though. It leads to higher resource consumption, which is more costly and inefficient for the environment. Another issue is the slow build and server tests execution. The microservice architecture helps to reduce those times, allowing each microservice's actions to run in parallel.

These issues motivate the need to migrate Artemis to microservices architecture. The digital transformation of Artemis from monolithic architecture can solve the above problems. Due to the distributed nature of microservices, developers can build and deploy them independently by fully automated

deployment pipelines. This procedure will accelerate the software development life-cycle of Artemis, reduce its downtime, and additionally make the overall Learning Management System (LMS) more fault-tolerant [6].

III. Objectives

The migration of Artemis consists of two phases. The first is to set the foundation of the new microservice architecture by extracting microservices from the monolith using the shared database pattern. The second phase aims to migrate the shared database to database-per-service.

The main objectives of the first phase includes the extraction of two microservices. Since there is a high coupling in the database usage, in order to minimize the migration risk, Artemis continues using the shared database pattern. The main goal is to set the microservices boundaries and to define a migration process to use in the microservice extraction. Another important part of the first phase is the Kubernetes deployment. Kubernetes helps to automate deployments, scale, and manage the applications. Moreover, it helps to achieve high availability thanks to its ability to automatically restart failed containers and automatic scaling features [7].

In this paper, we focus on the second phase of the migration. It aims to migrate the shared database into a database per microservice. The shared database is a limitation of the new architecture. Decomposing the database layer into many smaller databases per microservice is a time-consuming and critical process [8]. It requires the semantic distribution of the repository to different microservices domains, as well as the mechanism for handling the distributed transactions to keep the data consistent [9]. We point out the necessary steps and the concept of successive migration in section IV.

In the next phase of the migration process, we want to address another limitation, which is the increased complexity of the local development of the application, debugging and the development of integration tests. The deployment of the application is also more complex, it requires knowledge of Kubernetes orchestration system, its resources and various configuration options. This has a steep learning curve and can overwhelm developers, especially at the beginning of the learning process [10]. We plan to counteract the challenges by using the Architect framework for building dependable distributed systems.

IV. Method

In the next sections, we describe the methods that will help us eliminate the challenges from the first migration phase. We focus primarily on the concept of incremental database migration, highlighting the resulting challenges, and then how the **Architect** framework can help solve them.

A monolithic application architecture uses the shared database pattern for the implementation of data layer, which takes responsibility for the persistence of the application data. The migration to microservices also changes the paradigm of storage responsibility. Thus, the aspect of information hiding becomes important because several microservices access the data from this point instead of one monolithic application [2], [11].

Furthermore, the focus of the business logic within a subsystem implementation is on high cohesion, which implies that a particular microservice must have the own associated data storage. The use of shared database pattern violates this principle in a microservice environment, so developers should continue to use it only when the provided data is read-only or static [2], [11].

To ensure information hiding, data ownership and high cohesion, the microservices encapsulate their data and make the methods available for data access via their own Application Programming Interfaces (APIs). Therefore, it is necessary to split the monolith not only at the application level but also at the data level to ensure these properties.

While analyzing possible approaches to database migration, we found that the move from a shared database to database-per-microservice pattern is difficult to achieve via a big-bang approach. Each shared database has its specifics that developers should consider during the migration. For example, the database engine type plays an important role: it can be a relational or non-relational database. The number of tables, as well as the primary-foreign key relationships, can influence the complexity of the migration process. If developers can group the database tables, which contain many interconnections, by a specific bounded context - they can easily extract and assign them to a specific microservice; otherwise the migration can be critical for the existing production system, as they must separate already persisted data as well. In addition, the number and frequency of the data requests come into play, as well as the general load on the server node, which hosts the database.

To meet all these requirements, we analyzed different database migration patterns and tried to categorize them, building a taxonomy. This taxonomy allows us to design a successive migration concept using a decision process based on certain prerequisites of a monolithic database to provide support to the software architects or database specialists.

We design the decision-making process using the Unified Modeling Language (UML) activity diagram, putting together a sequence of different patterns based on the requirements of a current system which we want to use to migrate the monolithic database [12]. The goal of the migration process is to decompose this shared database to the database-per-microservice pattern.

The following path for the case study in Figure 1 shows the patterns' selection in green color using the directed arrows:

(1) Repository per Bounded Context → (2) Database per Bounded Context → (3) Monolith Data Access Layer → (4) Database Wrapping Service → (5) Change Data Ownership → (6) Data Synchronization → (7) Tracer Write.

The patterns result chain consists of 7 steps and represents a pure recommendation to reach the final result incrementally. Artemis Learning Management System (LMS) is a system that is already in productive use, so we do not recommend using the

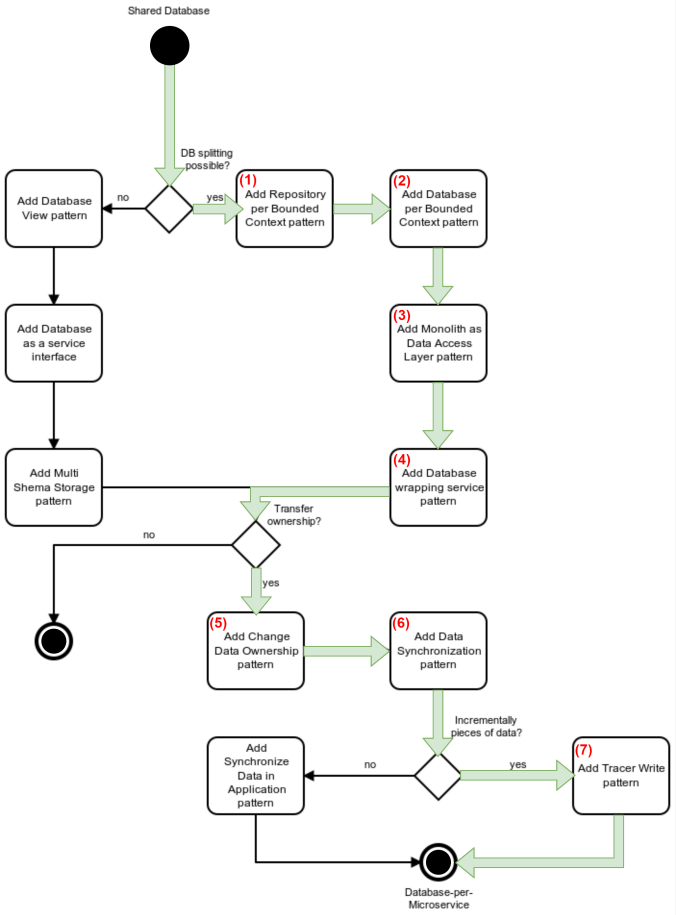

Fig. 1. UML activity diagram describing the path from a shared database to database-per-microservice pattern.

big-bang approach for such type of systems. The conditional nodes in the diagram consider the properties we defined before: information hiding, data ownership and high cohesion. The initial node stands for the shared database, which we have as input for the second phase after the first migration phase. The final node of the selected path symbolizes the desired migration result - database-per-microservice integration. For the simplicity reasons, we adapted the activity diagram for the Artemis case study, as we designed many more conditions nodes, which are not relevant in the given context.

First, there is the need to check if database decomposition is possible, which is the case for Artemis LMS. The next action is to chain a sequence of patterns to execute the first preparatory migration steps **(1)-(4)** for database decomposition. These steps ensure the information hiding property in the future decomposed system. Then, using pattern **(5)** follows the transfer of data ownership to the previously extracted microservices and finally with pattern **(6)-(7)**, the decomposition finishes by synchronizing the data to the new migrated databases. The last two steps ensure the high cohesion of the decomposed system.

In this section, we presented a concept for a step-by-step migration of the shared database to a database-per-microservice pattern. The distributed database system ensures important properties in a microservice architecture: information hiding, high cohesion and data ownership. However, it also introduces a new challenge that leads to eventual consistency - the distributed transactions. In the next section, we will go into this topic in detail and show possible ways how developers can solve this challenge by integrating Saga pattern.

A. Distributed Transactions using Saga pattern

The Saga pattern manages distributed transactions that span over several microservices. It can be implemented as event choreography, where a microservice that has executed a transaction, publishes an event to which other microservices subscribe and execute their own transactions [13]. This flow continues until no microservice publishes an event. Another way to implement the Saga pattern is by orchestration where a central coordinator listens to the events coming from the microservices and triggers other microservices' events [13].

Artemis also requires handling of distributed transactions. The plan for Artemis is to use the orchestrator implementation approach as it reduces the risk of introducing cyclic dependencies, which could happen in event choreography. Moreover, in orchestration, all the logic is in one place, which makes the process easier to change and trace [13].

An example for a distributed transaction in Artemis is deletion of a course. Each course has many components – exercises, lectures, students' submissions, etc. Courses are handled by the Artemis application server, while lectures – by the Lecture microservice. Therefore, this distributed transaction spans over two microservices. When an administrator wants to remove a course, the system should delete the course and its related components. In case of errors, all should remain consistent in the database.

Figure 2 describes Artemis' overall architecture. It includes Single Page Application (SPA) client application developed with the Angular Framework[1], API Gateway and two microservices—the User Management microservice and the Lecture microservice, and the Artemis application server.

In order to delete a course, first, the Lecture microservice deletes all lectures related to a given course, then the Artemis application server takes over to delete the rest of the course-specific components and the course itself. The so-called Saga Execution Coordinator [13] is responsible for the coordination and communication between the individual microservices. The coordinator in this case uses the Camunda Framework[2], which is a process orchestration tool that can be easily integrated into a cloud-native application thanks to its extensive Representational state transfer (REST) API and the ability to containerize it with Docker[3].

A specially developed workflow modelled in Business Processing Model Notation (BPMN)[4] triggers the lecture deletion process. The Camunda Database persists this workflow and controls its execution by providing REST API. It triggers each

[1]https://angular.io/
[2]https://camunda.com/
[3]https://docs.docker.com/
[4]https://www.bpmn.org/

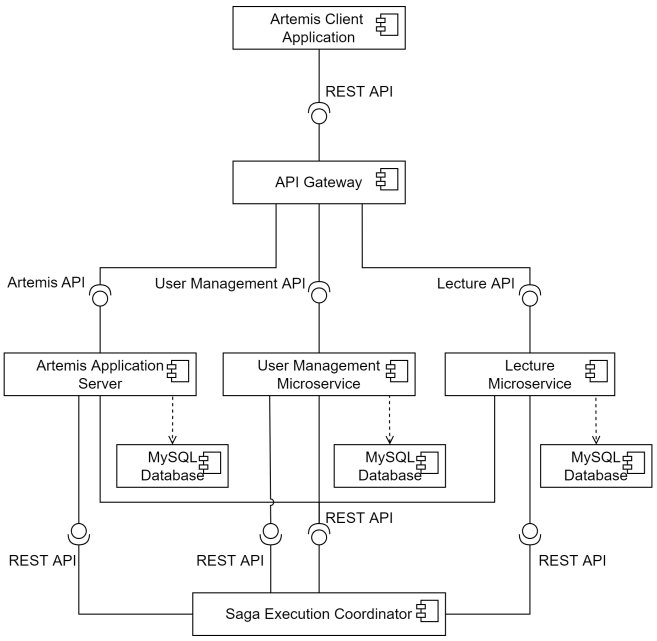

Fig. 2. UML component diagram of Artemis distributed transactions coordination using Saga pattern. The Saga Execution Coordinator which uses the Camunda Framework manages the distributed transactions and communicates with the microservices through REST API.

action part of the distributed transactions and the compensating actions if needed. BPMN is similar to a UML activity diagram[5]. The Camunda Engine provides persisted information about running workflows, their current status, can intercept and forward exceptions and provides the history of executed workflows.

The individual microservices have their databases[6], they use the background workers to pull the status of the workflow, to avoid double execution through action blocks and to send messages for the workflow continuation. The workflow must be modelled in such a way that the workflow actions not only execute the database transaction if successful, but also start the compensating transaction if an exception occurs, to keep the data consistent. There are many use cases which can throw an exception. For example, the lecture microservice deleted the related to a course lectures, but the course deletion fails. This potentially leads to an inconsistent data state that the compensating action can restore [9].

In the next section, we describe the **Architect** framework. We explain its features, semantic model and also the system design. Then we introduce the overall project generation process, from when the user uploads the description of Domain Specific Language (DSL)-based architecture to the **Architect** engine to deployment in the Kubernetes cluster.

B. Architect: A Framework for Building Dependable Systems

The microservices have not only advantages but pose many challenges in local development, deployment complexity, development of cross-cutting domain integration tests. The migration of the monolithic database to database-per-microservice pattern adds another challenge with distributed transactions handling [9]. Thus, the overall complexity increases considerably due to the division of the monolith into small autonomous units. Therefore, we propose DSL-based solution to support software architects and developers in dealing with such challenges [14], [15]. The solution is an open-source framework which leverages an external DSL called **Architect** developed for this purpose. **Architect** helps to integrate the design patterns in greenfield projects, to reduce the complexity by automatically generating the required architecture[7].

Figure 3 shows the current **Architect** semantic model, that contains 6 different domains which help to describe a dependable software architecture. In the next paragraphs, we describe the role of these domains in the architecture of the framework in detail. The semantic model represents the in-memory object model, that the DSL describes and thus is the library, that the Architect Domain Language (ADL) populates [14].

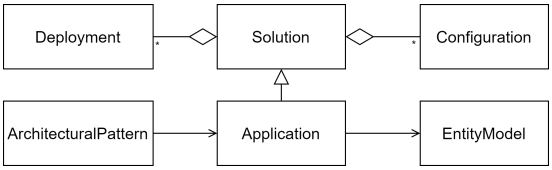

Fig. 3. Architect semantic model. It contains 6 domains with different purposes that together define a distributed software architecture.

1) Application: The domain allows the defining of different application types. They form the core of the generated solution and provide the possibilities for communication and data persistence through the definition of endpoints and their internal databases. We distinguish between 5 different types of applications that the user can use to build a state-of-the-art software architecture. These include client application, which can be either web-based SPA, mobile or desktop application; gateway, which is responsible for delegating the client requests to the services; microservice with database and service layers, constructed internally using ONION architecture [16]; and event bus to enable asynchronous communication between the individual services.

2) Solution: The domain defines the project solution structure, by providing the information about folder hierarchy and applications. Furthermore, it has information about the git repository, which will be responsible for the versioning of the project.

3) Configuration: With help of this domain user can define the application and solution configurations, including the folder structure, repository name, the list of different applications, their configuration and deployment resources.

4) Deployments: In this domain, the user describes the various resources that the **Architect** generates during the creation of the project to enable simple deployment in a cloud-native

[5]https://www.uml-diagrams.org/activity-diagrams-reference.html
[6]https://microservices.io/patterns/data/database-per-service.html
[7]https://docs.microsoft.com/en-us/azure/architecture/patterns/

environment. These include the docker and docker-compose files, various Kubernetes resources such as Persistent Volume Claim (PVC), Workloads, Services, Ingress Controllers, as well as Secrets and Configmaps. Describing these resources is very intuitive with **Architect**, but requires an understanding of Kubernetes and Docker concepts [7], [10].

5) Entity Model: This semantic model component adds the persistence layer support. The user can specify a Database (DB) engine, as well as multiple DB entities, including the relationships between the individual tables. Currently, **Architect** supports only relational databases like MySQL and PostgreSQL [17]. The individual DB entities can contain multiple fields of different data types. This flexible construct allows the definition of a basic DB structure that the responsible microservice application can use to persist the relevant data.

6) Architectural Patterns: The basic idea behind the development of **Architect** framework is to reduce the complexity of a distributed system, which can consist of multiple services that communicate with each other and therefore need support in complex scenarios. The integration of different microservice patterns should make this possible. **Architect** in its current development state can assist software architects by integration of Saga pattern[8], required for handling of distributed transactions, to ensure the data consistency. Architect modular system design, which utilizes microservice architecture and its flexible semantic model, allow adding more patterns with less effort.

The Figure 4 shows the process of the solution generation using **Architect** Engine, that contains client and server components and provides the complete E2E solution, including the deployment of the generated artifacts into the Kubernetes orchestrated cluster. The frameworks' server architecture leverages microservices, each of them contributing to the overall project generation process. We implemented an online DSL editor using Angular SPA framework to support users uploading their DSL-based architecture descriptions. In the following paragraph, we describe the workflow that illustrates the use of **Architect** and generation of a distributed E2E solution.

First, the user generates and imports the architecture description created with ADL Editor into **Architect Engine** server, which is responsible for the recognition of ADL, its transformation to semantic model, generation of project solution, Continuous Integration, Continuous Delivery (CICD) to the Kubernetes cluster and finally user notification. The **Architect Engine** contains multiple microservices responsible for these different tasks. **Architect Compiler** uses a grammar and parser created with ANother Tool for Language Recognition (ANTLR)[9] to recognize the imported ADL definition, transforms this meta information into a Data Transfer Object (DTO) structure based on the **Architect** semantic model and notifies the **Architect Generator** microservice via an event bus that it can proceed with project generation.

After an internal validation, **Architect Generator** persists the DTO structure in the internal DB, and then begins with the creation of the project structure using Yeoman scaffolding tool[10]. Once this process is complete, the **Architect Supplier** gets notification, begins the upload of the generated artifacts to the repository or local storage and their deployment to the Kubernetes cluster. **Architect Generator** provides all the necessary Kubernetes resources for deployment, as well as docker-compose files for the local development and debugging. After the process is complete, the system sends the notification to the user, who can then monitor the application state in the cluster.

Architect offers the ADL edit tool and engine, which allow the generation of a new project with integration of microservice patterns. The prototype implementation of Saga pattern significantly reduces the complexity of the required software solution, but we can only use it in a greenfield project. For this reason, we plan to use **Architect** currently as a template for the state-of-the-art implementation of orchestrated Saga pattern in the Artemis LMS case study. We adopt the business logic of BPMN workflows combined with the power of a Saga Execution Coordinator [13]. This approach allows us to build a sustainable solution that solves the challenge with distributed transactions and reduce significantly the complexity of the patterns' integration.

V. Results

The first phase of the migration to microservices included the extraction of two microservices from the Artemis monolith, resulting in three microservices applications which set the foundation of the new microservice architecture. As a result, we can independently scale each of the new microservices. This phase did not include splitting the database and migration to database-per-microservice, though. The first phase was a proof of concept for the possibility to migrate the Artemis monolith application to microservices.

After the success of the first phase, follows the second phase where the focus is on splitting the database, which suggests incremental database migration. The migration concept consists of several steps. First comes the need to assign the database repositories to each database, making sure that other microservices do not have direct access to databases assigned to other microservices and connect only to their own database. The next step is to change the data ownership and transfer the data itself by making sure that the data is consistent.

The implementation of the database-per-microservice pattern poses the challenge of managing distributed transactions, which the Saga pattern solves. The pattern helps to manage the distributed transactions by producing events that the Camunda Engine orchestrates.

The **Architect** framework helps to reduce the complexity of the integration of the Saga pattern. Artemis uses it as a template to solve the challenge with distributed transactions which is one of the most important challenges [18].

[8]https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga/

[9]https://www.antlr.org/, https://tomassetti.me/antlr-mega-tutorial/

[10]https://yeoman.io/, https://tomassetti.me/code-generation/

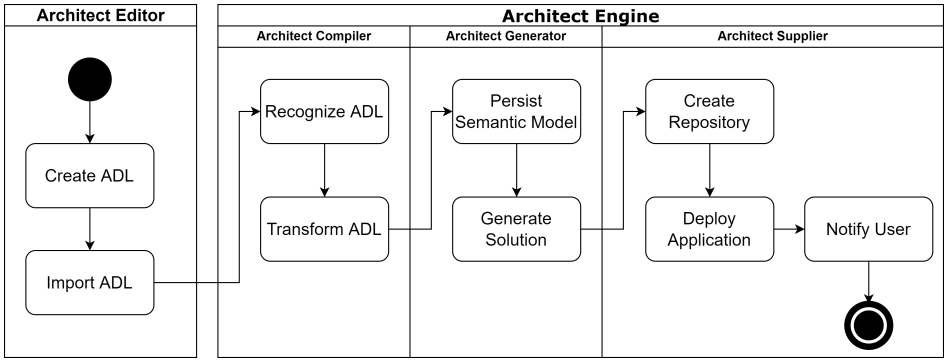

Fig. 4. The Architect Engine activity diagram. It explains the Architect microservice-based process, which provides the complete E2E solution, including the deployment of the generated solution artifacts into the Kubernetes orchestrated cluster.

Architect also makes the deployment of the new architecture easier by generating Kubernetes and docker-compose files providing the E2E solution which developers can easily run and deploy in cloud [19]. Moreover, Kubernetes' self-healing property by monitoring and automatically restarting failed containers help for the high availability of the system [20].

VI. Conclusion

In this paper, we introduced the **Architect** framework that supports the migration process and enables the creation of a dependable architecture for the distributed applications. However, the current state of the framework has much improvement potential for the existing projects.

For example, developers can only use **Architect** to create projects from scratch, so it does not provide a software evolution approach. For this reason, the case study could use **Architect** only as a template for a state-of-the-art implementation. A software product is constantly evolving, developers enhance the features set and fix bugs to improve the overall software quality. Such development activities always address recurring problems or introduce new challenges that require the use of design patterns. Thus, there is the need to develop an automated mechanism for integrating the new design patterns into the existing software solution.

Furthermore, Architect currently supports only relational databases, which is an improvement point for later development because, especially in the serverless area, developers often use non-relational alternatives. In addition, Architect generates dotnet-based services, which can severely limit developers in their choice of technology stack. Thus, an extension by adding support for new programming languages and frameworks will be beneficial. In conclusion, **Architect** helps to improve the code quality, saves the development time and shows the potential for making a significant contribution in the software evolution research area by approaching the migration of the existing projects to the dependable architecture.

References

[1] Joseph Ingeno. *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd, 2018.

[2] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[3] J Fritzsch, J Bogner, S Wagner, and A Zimmermann. Microservices migration in industry: Intentions, strategies, and challenges.

[4] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4, 10 2017.

[5] Stephan Krusche and Andreas Seitz. Artemis: An automatic assessment management system for interactive learning. SIGCSE '18, 2018.

[6] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7, 2019.

[7] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 970–973, 2018.

[8] Rajiv Srivastava. *Cloud Native Microservices with Spring and Kubernetes: Design and Build Modern Cloud Native Applications using Spring and Kubernetes*. BPB Publications, 2021.

[9] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, page 249–259. Association for Computing Machinery.

[10] Gigi Sayfan. *Mastering Kubernetes*. Packt Publishing Ltd, 2017.

[11] Scott W Ambler and Pramod J Sadalage. *Refactoring databases: Evolutionary database design*. Pearson Education, 2006.

[12] Bernd Bruegge and Allen H Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 2010.

[13] Chaitanya K Rudrabhatla. Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 9(8):18–22, 2018.

[14] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[15] Ernst Oberortner, Uwe Zdun, and Schahram Dustdar. Domain-specific languages for service-oriented architectures: An explorative study. In *European Conference on a Service-Based Internet*, pages 159–170. Springer, 2008.

[16] M. Ehsan Khalil, Kamran Ghani, and Wajeeha Khalil. Onion architecture: A new approach for xaas (every-thing-as- a service) based virtual collaborations. *2016 13th Learning and Technology Conference, L and T 2016*, pages 13–19, 9 2016.

[17] A comparison of a graph database and a relational database: A data provenance perspective. *Proceedings of the Annual Southeast Conference*, 2010.

[18] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. 09 2018.

[19] Salvatore Augusto Maisto, Beniamino Di Martino, and Stefania Nacchia. From monolith to cloud architecture using semi-automated microservices modernization, 2020.

[20] M. Luksa. *Kubernetes in Action*. Manning, 2018.