Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report

Chen-Yuan Fan
Department of Computer Science and Engineering
National Taiwan Ocean University
Taiwan
b4456609@gmail.com

Shang-Pin Ma
Department of Computer Science and Engineering
National Taiwan Ocean University
Taiwan
albert@ntou.edu.tw

***Abstract*—The microservice architecture (MSA) is an emerging cloud software system, which provides fine-grained, self-contained service components (microservices) used in the construction of complex software systems. DevOps techniques are commonly used to automate the process of development and operation through continuous integration and continuous deployment. Monitoring software systems created by DevOps, makes it possible for MSA to obtain the feedback necessary to improve the system quickly and easily. Nonetheless, systematic, SDLC-driven methods (SDLC: software development life cycle) are lacking to facilitate the migration of software systems from a traditional monolithic architecture to MSA. Therefore, this paper proposes a migration process based on SDLC, including all of the methods and tools required during design, development, and implementation. The mobile application, EasyLearn, was used as an illustrative example to demonstrate the efficacy of the proposed migration process. We believe that this paper could provide valuable references for other development teams seeking to facilitate the migration of existing applications to MSA.**

Keywords—microservice; microservice architecture; migration; software development life cycle

I. INTRODUCTION

The rapid development of the internet has greatly increased the number of requests for services on demand, and the growing complexity of software systems makes them increasingly difficult to manage. Traditional enterprise applications are divided into the front-end UI (user interface), service-side logic components, and database. Front-end UI components run on user devices, such as web pages or mobile-side interfaces. Server-side logic components run on a server or in the cloud. The back-end database hosts application data. Server-side components work in conjunction with the database to handle requests issued by users. The above application architecture is referred to as monolithic. Monolithic architecture needs to struggle with handling cases in which the number of users exceeds the capacity of the server. Furthermore, monolithic architectures are hard to manage and maintain due to a lack of mechanisms aimed at modularization.

Microservice architectures (MSAs) are an emerging cloud-based software system, which differ fundamentally from monolithic systems. MSA [1-5] provides intermediate building blocks, referred to as microservices (heretofore also referred to as services), to handle specific functions. Each service is self-contained within an independent life cycle, utilizing lightweight communication mechanisms to work in conjunction with other services. These services are easily implemented using different programming languages, frameworks, platforms, and databases. MSA is commonly integrated with DevOps techniques to automate system development via continuous integration and continuous deployment [6]. The monitoring of applications also makes it possible for MSA to obtain feedback to facilitate system improvements, thereby shortening the development cycle. Several enterprises, such as Netflix and Amazon, have adopted the microservice architecture to build scalable, easily maintainable software systems.

For the sake of scalability, an MSA should be integrated with Service Discovery, API Gateway, and Circuit Breaker. Service discovery provides a mechanism for the registration of microservices. This makes the services addressable, such that they are able to communicate with each other using an IP and port [7]. API Gateway is an interface between the client and microservice system, which allows users to invoke the services they require without the need for any knowledge pertaining to the structure of the system. Circuit Breaker is an error handling mechanism to increase the resiliency of the system. Various automation tools are required for the application of MSA. A pipeline [8] breaks down the software delivery process into stages, including build automation, continuous integration [6], test automation, and deployment automation. The use of containers [9] in the pipeline facilitates deployment and portability to accelerate the delivery of software systems to users. In large-scale microservice systems, centralized logs, metrics, and feedback systems can be used to facilitate management and on-going development.

At present, although there are a lot of books and papers to discuss tools, patterns, and practices on MSA, systematic and SDLC-driven methods (SDLC: software development lifecycle) are still lacking to facilitate the migration of software systems from a traditional monolithic architecture to MSA. Therefore, in this study, we adopted a number of existing tools in the migration process based on SDLC. The proposed migration process includes mechanisms for design, development, and operations. We believe that the proposed method is a valuable reference for the development of similar

978-1-5386-1999-5/17 $31.00 © 2017 IEEE
DOI 10.1109/AIMS.2017.23

applications aimed at facilitating migration from a monolithic architecture to a microservice architecture. The mobile application, EasyLearn, was used as an illustrative example to demonstrate the efficacy of the proposed migration system.

The remainder of this paper is organized as follows: Section II describes an illustrative example, EasyLearn, and outlines the proposed approach to software system migration from a monolithic architecture to MSA. Finally, Section III presents our experiences obtained using the proposed migration system and draws a number of conclusions.

II. Proposed Migration Process

A. Illustrative Example: EasyLearn

EasyLearn is a mobile application used to facilitate creation, sharing, reading, discussion, and updating of condensed articles (also referred to as a condensed article as a "Pack" in the system). We designed the system architecture for EasyLearn using three subsystems, Android App, Web App, and RESTful services. EasyLearn was implemented as a monolithic application published to Google Play. During system operation, we encountered two problems:

1. Attempting to change the database schema needs by adding new features or modifying the existing functionality required that the related code required be modified as well. Despite our design of well-defined interfaces, coupling between modules remained high.
2. The synchronization service was forced to handle a great deal of string processing, which greatly increased the consumption of processor and memory resources. An increase in the number of requests for services led to longer response times, eventually to the point at which the functionality of app was compromised.

In an attempt to overcome these difficulties, we migrated EasyLearn from the original monolithic architecture to a microservices system. It was hoped that this might also enhance the scalability and maintainability of the system.

B. Process Overview

Migrating a software system from a monolithic architecture to a microservice architecture is not a trivial task. We opted for a gradual (iterative) approach to migration in order to maintain the stability of the overall system. This also made it possible to deal with problems that occurred during migration, such as incorrect service identification, inappropriate assignment of responsibility, and interface mismatch, as they were encountered. In this section, we propose a systematic approach to (shown in Figure 1) migration of software systems from a monolithic architecture to a microservice architecture. This process begins with analysis of the internal system architecture using Domain-Driven Design (DDD) [10] to extract candidate microservices from the original system. The next step involves determining whether the database schema is consistent with the candidate microservices and the filtering out of inappropriate candidates. This is followed by the extraction and organization of code related to the service candidates using Java. The Java interface is treated as a temporary service interface in order to ease communications between services. Following service code extraction, the operator selects a communications protocol, data format, and microservice framework. The operator must also determine whether the current database is appropriate for the new system and make adjustments accordingly. Finally, the Java interfaces are transformed into actual service interfaces, such as REST or MQTT, whereupon service invocations are implemented to enable communication between the various services.

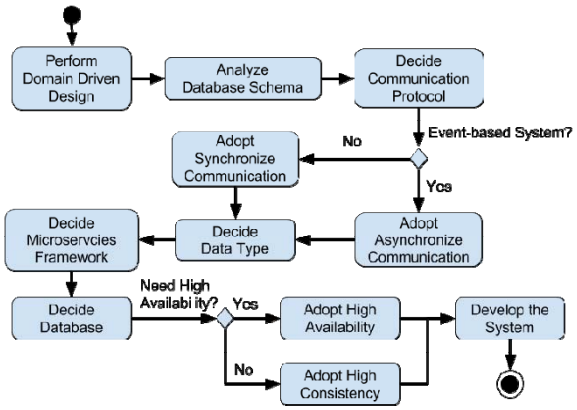

Figure 1 Proposed migration process

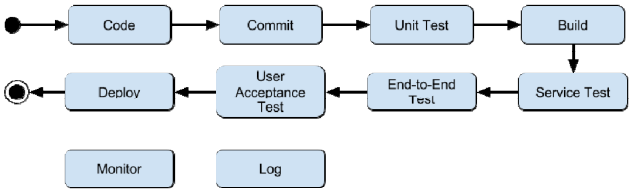

Figure 2 Proposed development process

In the development process (shown in Figure 2), developers must push the code to the version control system in order to trigger subsequent automated processes, such as continuous integrating, testing, and deployment. Unit testing should be conducted first, the same as normal software projects. Runnable artifacts are then built and placed within a container [9] image to be tested by service test and end-to-end test. Note that service test takes into account the consumer-driven contract between a service provider and its service consumer to ensure successful service invocation. Following acceptance testing of the entire system, it is ready to be deployed to the production environment. Throughout these processes, Log and Metrics analysis tools can be used to manage and analyze the logs centrally and collect data during service execution.

C. System Analysis for the Identification of Microservice Candidates

Two analysis methods are used in the migration of a monolithic system into a microservice system. We first use DDD to analyze various aspects of the system architecture. The bounded context (BC) of the analysis results provides candidates for the splitting of services. There is no canonical way to split services, which can vary considerably with regard

to system requirements, team size, and change requirements. The above-mentioned DDD is a method used in the analysis of complex systems, with the following characteristics: 1) Each project is for a specific domain; 2) Complex systems are composed of a number of domain modules; and 3) Domain problems are analyzed by developers in conjunction with domain professionals. Each microservice is designed as an independently operable component, and DDD-based analysis is used to facilitate the extraction of domain-specific services. The design principles of DDD approach promote the design of low-coupling microservices.

The second method involves analysis of the database structure. The repository layer is generally used by a number of modules, which results in a high degree of coupling between modules. Foreign keys can be marked as microservice candidates. In a microservice system, it is preferred that each service uses a discrete database; i.e., does not share a database schema with other services. This reduces coupling between services; however, splitting data into separate repositories may cause data inconsistency.

Many factors determine the extraction of service candidates from existing modules, such as system requirements and the nature of the database. For example, a high load module could possibly be wrapped as an individual service to allow further scaling. The high coupling associated with excessive communication between services can be alleviated by merging services.

In the above method, microservice candidates can be split using a package or module in a particular programming language, such as Java or node.js. Next, it is necessary to determine whether the current interfaces of microservice candidates (i.e. package or module) are appropriate, and whether communication between service candidates is too cumbersome. Only after confirming that the system works well do we proceed to the next stage.

In the case of EasyLearn, we first use DDD analysis to obtain the following microservice candidates: Sync Service, User Service, Pack Service and Note Service (shown Figure 3). We decided to split the note/comment module to an independent service, rather than include it in the Pack service, due to the high likelihood of frequent updates and the need for scalability. As shown in Figure 4, the database schema is divided into three parts, similar to the results of DDD.

Sync Service provides data synchronization and an interface for front-end systems. User Service is in charge of user profiles, folders, bookmarks and other information, using the user-related database structure indicated by the solid line in Figure 4. Pack Service is in charge of condensed articles and their various versions, the database structure of which is indicated by the dashed line in Figure 4. Note Service is in charge of notes, the database structure of which is indicated by the double line in Figure 4.

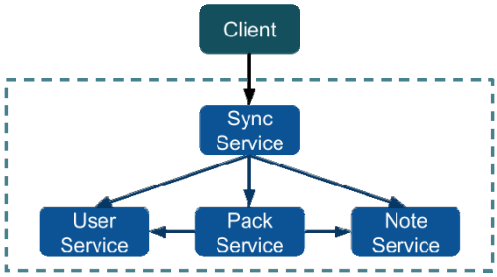

Figure 3 System architecture based on domain driven design

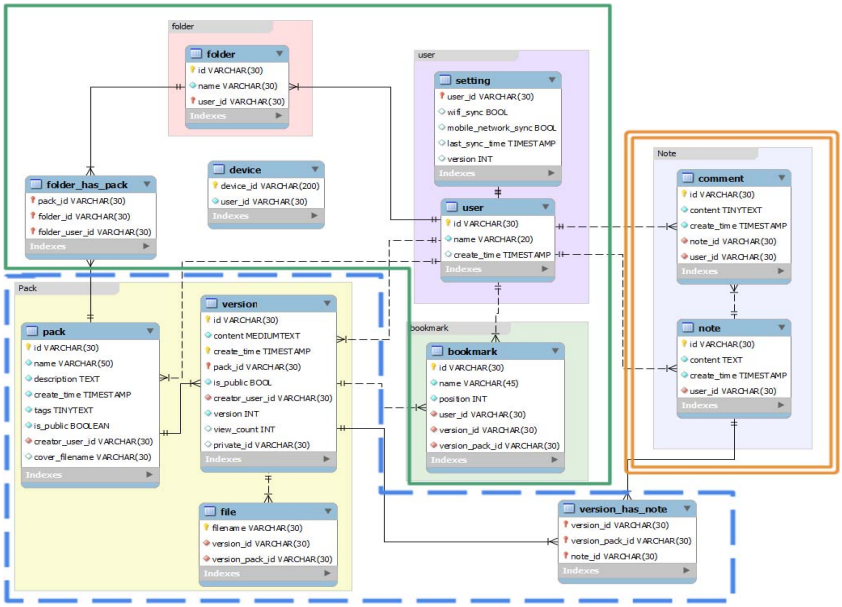

Figure 4 Database structure after database analysis

III. DISCUSSIONS AND CONCLUDING REMARKS

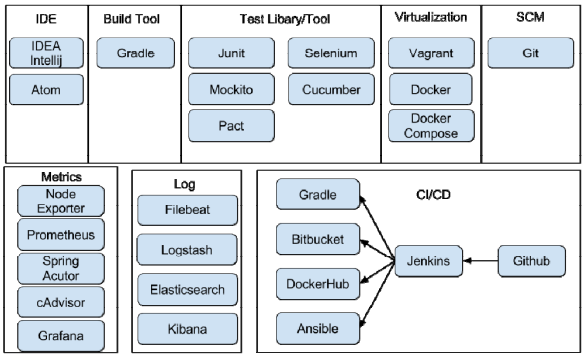

Figure 5 Tools used in the migration and operation

In this paper, we used EasyLearn as an example to illustrate the migration from a monolithic to a microservice architecture (MSA) and address the issues encountered in this process. In an MSA, each database server is managed by only one microservice to ensure that the refactoring of the database does not affect the microservice interfaces, thereby ensuring that other services will not be affected. This makes the system more flexible and scalable. Scale up by preparing multiple identical services to balance loads is easy, particularly for high-loading services. In the operation of the microservice system, microservices logs and metrics are used to collect data, which can then be visualized in charts to facilitate analyzing the health of the system. The tools used for these processes are listed in Figure 5.

Based on our experience in migrating the EasyLearn system, we draw the following conclusions concerning microservice systems:

- Automated: When a new version of a microservice is built, regardless of whether it is a minor enhancement or a major overhaul, integration and deployment in the pipeline requires only re-execution of unit test cases and service test cases. Developers need only focus on the source code of microservices, because testing and deployment are automated. Furthermore, centralized management of logs and metrics provides valuable information, which can be used to solve problems and improve the system.
- Specialized and simple: Microservices are designed to handle problems in a single domain, and the interface between microservices is clear. Single microservices use only one database schema, which brings the benefits of easy refactoring, migration, and updates to the database. It is also relatively easy to develop, and test a microservice on a local machine.
- Fault Tolerance: Microservices are designed for fault tolerance. When other services fail to function properly, it is dealt with by an error handling mechanism, rather than rendering the entire system unusable. For example, when one of the microservices is not working, the system automatically routes the request to the healthy one. Even if there is no health microservice, the user can still add a condensed article.

We also identified two disadvantages of MSA:

- 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. This includes monitoring tools, server environment, automatic deployment tools, and automatic integration.
- Using more resources: Microservices use multiple tools to achieve architectural flexibility, such as Service Discovery and API Gateway. This consumes more resources and increases the complexity of the system.

In summary, this paper introduces a process to facilitate migration from a monolithic architecture to a microservice architecture. We believe that the proposed approach and case study could provide valuable references for other development teams seeking to migrate existing applications to MSA. Our future research plan involves the design of quantitative experiments to further illustrate the efficacy of the proposed migration method based on software metrics.

ACKNOWLEDGMENT

This research was sponsored by Ministry of Science and Technology in Taiwan under the grant MOST 105-2221-E-019-054-MY3.

REFERENCES

[1] B. Familiar, "What Is a Microservice?," in *Microservices, IoT, and Azure*, ed: Springer, 2015, pp. 9-19.
[2] S. Newman, *Building Microservices*: " O'Reilly Media, Inc.", 2015.
[3] J. Lewis and M. Fowler, "Microservices," ed: Recuperado de: http://martinfowler.com/articles/microservices.html, 2014.
[4] C. Richardson, "Microservices: Decomposing applications for deployability and scalability," ed: InfoQ, 2014.
[5] J. Lewis and M. Fowler. (2014). *Microservices - a definition of this new architectural term*. Available: https://martinfowler.com/articles/microservices.html
[6] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, p. 122, 2006.
[7] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," *arXiv preprint arXiv:1609.05830*, 2016.
[8] M. Soni, "End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery," in *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015, pp. 85-89.
[9] H. Kang, M. Le, and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 202-211.
[10] E. Evans, *Domain-driven design: tackling complexity in the heart of software*: Addison-Wesley Professional, 2004.