JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 1285-1303 (2022)
DOI: 10.6688/JISE.202211_38(6).0010

Microservice Migration Using Strangler Fig Pattern and Domain-Driven Design*

SHANG-PIN MA[1,+], CHIA-YU LI[1],
WEN-TIN LEE[2] AND SHIN-JIE LEE[3]
[1]*Department of Computer Science and Engineering*
National Taiwan Ocean University
Keelung, 202 Taiwan
[2]*Department of Software Engineering and Management*
National Kaohsiung Normal University
Kaohsiung, 802 Taiwan
[3]*Department of Information Engineering and Computer Science*
National Cheng Kung University
Tainan, 701 Taiwan
E-mail: albert@ntou.edu.tw[+]

The microservice architecture (MSA), comprising multiple autonomous microservices, is easy to upscale, test, and maintain. Many enterprises are seeking to replace the monolithic architecture with MSA to enhance software quality; however, researchers have yet to develop a systematic approach to microservice migration. In the current study, we developed a microservice migration scheme based on the Strangler Fig pattern and Domain-Driven Design (DDD). The efficacy of the proposed scheme was evaluated in two cases studies, including the DataCustodian system of the Green Button project and the BBDP (Bridge between Doctors and Patients) system.

Keywords: microservice, architecture migration, strangler fig pattern, strangler fig application, domain-driven design

1. INTRODUCTION

Most legacy applications were established within a monolithic architecture; however, the accretion of technical updates, personnel transfers, requirement changes, and feature additions eventually develop into a "big mud ball" [1], which significantly reduces scalability and maintainability. Meanwhile, as the technology of AI (Artificial intelligence) and IoT (Internet of Things) is getting mature, combining AI and IoT services is challenging due to the diversity of the complicated IT systems [2]. The microservice architecture (MSA) [3] comprises multiple, autonomous services with their own business logic and databases to facilitate discrete deployment and testing. The modular nature of microservices makes MSA highly scalable and easy to maintain [4]. There are considerable advantages to switching over from monolithic to microservice-based applications [5] and integrating diverse and complex systems; however, a systematic process is not well established for the migration of microservices.

In 2004, Fowler proposed a software modernization strategy referred to as the Strangler Fig application [6]. The namesake of the application grows upward around a tree to

Received October 31, 2021; revised December 3, 2021; accepted December 31, 2021.
Communicated by Meng-Hsun Tsai.
* This research was sponsored by the Ministry of Science and Technology in Taiwan under grants MOST 108-2221-E-019-026-MY3 and MOST 110-2221-E-019-039-MY3.

reach the sunlight, whereupon the tree eventually dies, leaving only a tree-shaped vine. This is analogous to the refactoring approach adopted for the gradual rewriting of an existing system in the incremental migration of a monolithic architecture to MSA, rather than the "big-bang" migration that decomposes a monolith into interconnected microservices at once. Notably, the Strangler Fig pattern allows pausing and stopping the migration and still taking advantage of the newly-split microservices. Accordingly, we can improve the modularity and maintainability for the newly-split microservices, and enhance the performance of the remainder monolith since some modules are cut and converted into microservices. Besides, because refactoring a high-coupling or low-cohesive monolith system to rebuild interconnected microservices at once is very costly and risky, it is evident that the big-bang migration is impractical on many occasions.

The first thorny problem encountered in microservice migration involves splitting the system into candidate microservices. Multiple methods were devised to extract or identify microservices from a monolithic system, such as cutting a monolith based on the coupling criteria [7], identifying microservices from the business processes [8], aggregating service endpoints to form microservices using semantic reasoning for service interfaces [9], and extracting virtual abstract dataflow to retrieve microservices [10]. However, DDD (Domain-Driven Design) [11, 12] has been chiefly suggested for use and widely adopted when identifying or designing microservices [13-16]. DDD is a software development methodology involving a continual process of iterative improvements toward establishing complex system architecture. As mentioned by [17], DDD is suitable to deal with the complexity encountered in designing distributed systems, especially in large and complex domains. In the field of microservices, DDD can provide a systematic way to set an appropriate boundary for each service by breaking the domain into a series of bounded contexts. Therefore, in this research, we employed DDD to facilitate the decomposition of the original monolithic system into multiple parts in accordance with strategic and tactical objectives from the perspective of domain-specific business logic.

We combined the Strangler Fig pattern with DDD to automate processes involved in service identification, preparatory, service prioritization, service implementation, integration, testing, and the removal of legacy modules. The efficacy of the proposed scheme was evaluated in two cases studies, including the DataCustodian system of the Green Button project and the BBDP (Bridge between Doctors and Patients) system, from the viewpoint of QoS (Quality of Service).

The remainder of this paper is organized as follows: Section 2 introduces related work. Section 3 details the proposed migration approach. Section 4 presents the results of implementing the proposed approach on two existing systems. Section 5 discusses the design and the results of the evaluation experiments. Section 6 outlines conclusions and future work.

2. RELATED WORK

Knoche *et al.* [18] broke down the process of microservice migration into five stages: defining an external service facade, adapting the service facade, migrating clients to the service facade, establishing internal service facades, and finally replacing service implementations with microservices. The underlying objective is to develop well-defined, platform-independent interfaces based on the underlying bounded context. This process is

suitable for migrating large, complex software systems. In the book "Monolith to Microservices" [19], Newman detailed the methods used to migrate existing monolithic systems to microservices, including Strangler Fig Applications, Branch by Abstraction (BBA), Aggregate Exposing Monolith, Repository per Bounded Context, *etc.* These patterns provided a valuable resource in our current research.

There are numerous research efforts for microservice identification. Gysel *et al.* [7] proposed an approach to service decomposition based on 16 coupling criteria. The resulting Service Cutter uses a "nanoentity" as a unit by which to extract coupling information from software engineering artifacts for use in plotting an undirected, weighted graph to find and score densely connected clusters, and aggregate loosely-coupled and highly-cohesive services. Petrasch [20] proposed a model-based approach to the design and integration of microservice architectures using formal UML profiles to deal with specification gaps between microservices. Their approach extends UML component diagrams to model the bound context pattern in DDD and enable the modeling of microservices for subsequent model-to-model conversion and code generation. Baresi *et al.* [9] developed a microservice identification method based on the semantic reasoning for OpenAPI specifications. Their scheme involved matching concepts between a reference vocabulary and OpenAPI specifications, whereupon semantic similarity is calculated according to the number of co-occurrences to identify candidate microservices for microservice decomposition. Amiri [8] proposed a scheme to model a software system as a set of BPMN (Business Process Model and Notation) business processes with data object reads and writes for use in aggregating and clustering the structured dependency and object dependency values between two activities. The clusters are then identified as microservices. Chen *et al.* [10] developed a dataflow-driven decomposition algorithm to retrieve microservices, including three steps: (1) engineers and users construct a detailed dataflow diagram of the business logic; (2) the proposed algorithm combines the same operations with the same type of output data into a virtual abstract dataflow; and (3) the algorithm extracts modules from the virtual abstract dataflow to identify microservice candidates. Schmidt and Thiry [21] concluded that there are two main trends of studies for microservice identification in a systematic review they conducted. The first kind of method applied tracking systems to record dynamic behaviors for use in identifying candidate services, and the second kind of studies that identified microservices from established business models, such as requirement models, business processes, and data flows.

3. MMSD: MICROSERVICE MIGRATION

This section describes the proposed microservice migration scheme, referred to as Microservice Migration using the Strangler fig pattern and Domain-driven design (MMSD). Fig. 1 illustrates the process underlying MMSD. Note that the migrated service is referred to as the Strangler service. The Strangler application comprises multiple Strangler services used to replace the monolithic system. Note that the activities in light yellow are the core steps that will be fully discussed in this paper. The process includes the following steps:

1. First, analyze the monolith and divide it into bounded contexts as candidate services by using DDD.

2. Create a service registry and an API gateway.
3. Select a candidate service (*i.e.*, a strangler service) to be migrated based on the proposed method of service value calculation.
4. Construct the new service based on legacy code and register it on the service registry.
5. Develop integration glues in the monolith to link the new service using the API gateway.
6. Conduct API testing for the new service. If the testing is failed, we can return to the step of service construction to fix the defect; If the testing is passed, we can remove the legacy module from the monolith to realize the Strangler Fig pattern.
7. After the completion of the migration for the target service, we need to determine if the whole migration is completed. If the whole migration is not completed, we can return to the step of the selection and development of the next strangler service.

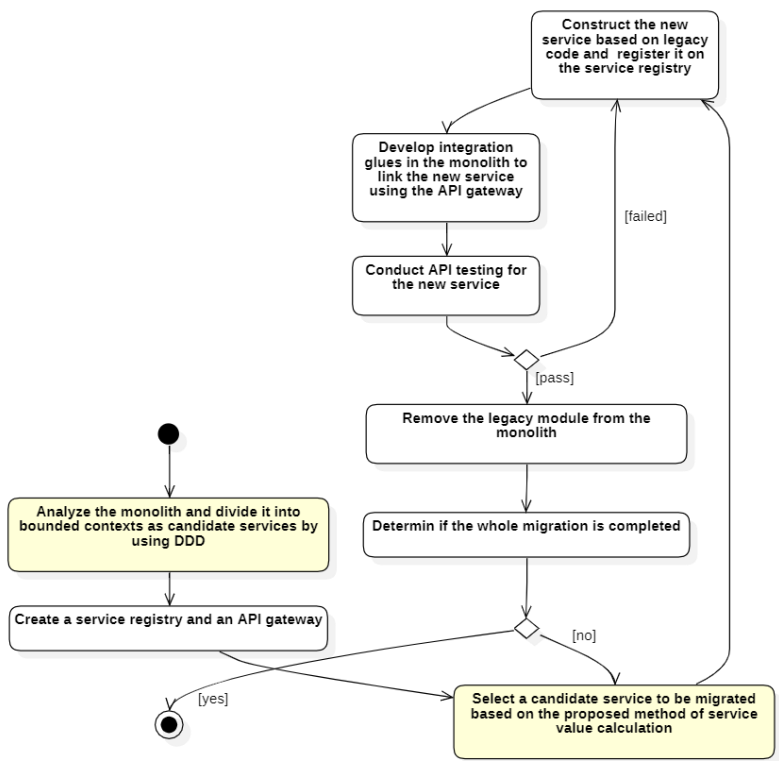


Fig. 1. Process of the MMSD approach.

3.1 Service Identification

As mentioned, DDD is used to identify microservices in this study. This process requires a ubiquitous language in which the terms and concepts of the business domain are clearly identified. The underlying objective is to balance technical and business issues and then use the ubiquitous language in subsequent conversations to enable the use of event storming, domain storytelling, or user story mapping to identify requirements, characterize

entity classes and their relationships, and build an abstract domain model for the system.

The first step in distilling the domain model involves strategic design. The domain is expressed as a problem space. Before finding a solution space, the domain is divided into multiple subdomains according to business logic, which is then classified into core domains, supporting subdomains, and generic subdomains.

Analysis of the problem space is then used to guide the definition of Bounded Contexts (system service boundaries) by which to describe the solution space corresponding to subdomains in the problem space. Bounded Context is usually defined in a common language in accordance with semantics and business capabilities. In this study, the Bounded Context is regarded as a candidate microservice to be migrated. Context mapping is also used to determine the means by which different Bounded Contexts interact with each other. In context mapping, downstream services depend on upstream services. Open Host Service/Published Language (OHS/PL) and Anti-Corruption Layer (ACL) are commonly used patterns, respectively, used for upstream services and downstream services. The OHS/PL pattern is used in the upstream services to facilitate the service collaboration, and the ACL pattern is applied in some downstream services to enable the conversion of data in accordance with the requirements of upstream services.

Tactical design is then used to distill the results of strategic design. The design patterns most commonly used for tactical design include entities, value objects, and aggregates, also referred to as domain objects. Ultimately, each Bounded Context is treated as a candidate microservice with a corresponding model.

Following the completion of domain-driven design, we recommend mapping the models and database tables (in the original system) within the candidate microservices (Bounded Contexts) based on the domain model. This step is essential to the conversion of existing modules into new microservices to be used by developers to evaluate the design. During mapping, unmatched models (in the original system) should be analyzed again for use in modifying the design.

MSA offers many advantages; however, the complexity of communications in container-based services imposes a heavy burden in terms of maintenance and operations. Thus, for the preparatory tasks of the MSA environment, automated management mechanisms are required to alleviate the burden in development and operation, and numerous new technologies applicable to MSA have been developed [22]. In the current study, we selected Eureka for service discovery and registration, and Spring Cloud Gateway for the management of API routing. We built a service registry (Eureka Server) and an API gateway (Spring Cloud Gateway) for deployment in a lightweight container (Docker).

3.2 Service Prioritization

The migration process requires clearly defined criteria by which to derive the order of services for extraction. It also requires a comparison of the expected benefits of the identified service to facilitate prioritization. Therefore, we reviewed the existing literature for the criteria of the scoring guidelines used to determine values for identified services (See Table 1).

We devised five criteria, including FM (Function to be Modified), PP (Performance Problem), CF (Core Function), RS (Related Service), and SS (Separate Service). Note that the curly brackets express a value set; namely, the score of the criterion for a target service

is set to one of the elements in the set; the square brackets indicate a range; namely, the score of the criterion for a target service is set to a value in the range.

We divide these criteria into three importance levels. FM is the highest prioritized criteria because the candidate service has been planned to be modified and is very suitable to be converted into a microservice. PP and CF are the second prioritized criteria, because the performance issue and the maintenance of core functionality should be paid considerable emphasis. RS and SS are the supporting criteria to determine appropriate services to be migrated by applying the Strangler Fig pattern. Based on the importance levels, we allocate the maximum scores of FM, PP, CF, RS, and SS to 40, 20, 20, 10, and 10, respectively, to represent their importance and let the sum of the maximum scores be 100. Besides, because the FM indicator is either fully satisfied or fully unsatisfied, its score will be zero or forty, not a value in a range.

After service value scores for all candidate services are calculated, the highest score represents the highest expected benefits, indicating that the corresponding service should be extracted first. Please refer to section 4 to see examples of service value calculation.

Table 1. Guidelines for the scoring of service value.

Criteria	Score	Rationales/Benefits	References
FM (Function to be Modified): The functionality involved in the service is about to be modified.	{0, 40}	Extraction of this type of service can eliminate the need to make changes to the monolithic system directly.	[23-25]
PP (Performance Problem): The service has performance problems**.**	[0, 20]	If the excessive consumption of resources by a candidate service affects the performance of other modules, then it should be extracted early.	[23, 24, 26]
CF (Core Function): The service involves core system-level functions**.**	[0, 20]	The extraction of services involving core functions brings added benefits, such as scalability and/or maintainability.	[25, 27]
RS (Related Service): The service is related to a service that has previously been migrated.	[0, 10]	Extracted services should be able to collaborate with other migrated services.	[23]
SS (Separate Service): The service is obviously distinct from the monolithic architecture.	[0, 10]	This kind of service is more easily extracted than are other candidates.	[25]

3.3 Service Implementation, Integration, and Testing

After identifying the services to be extracted, we recommend setting up a new microservice project on a version control system (VCS), such as GitHub or GitLab, before implementing Spring Cloud Gateway and the Eureka client.

The service should be implemented by adding models of the corresponding service based on the legacy code, creating controllers to define the service interface, and establishing a new database for that service. To facilitate collaboration between the original monolithic service and the new service (*i.e.*, the design of integration glue/adapter), we adopted the following strategies:

1. *Using REST API to query data:* The Aggregate Exposing Monolith pattern can be used to facilitate collaboration. The APIs required by the new service are designed and

provided by the original monolithic system, while the new service also provides APIs needed by the monolith. In other words, the monolith and the new service communicate with each other via API calls.

2. *Updating data in an event-driven manner:* Maintaining data consistency requires the triggering of updates by posting publish/subscribe events in the monolith and new service. We can leverage an event bus to enable subscription, un-subscription, and publishing events.

Microservice testing includes unit testing, integration testing, component testing, and end-to-end testing (E2E testing) [23]. We recommend conducting integration testing and component testing at the very minimum.

Finally, we suggest preparing a docker file to specify commands used to host the new microservice in a container and configure a continuous integration tool (such as Jenkins or Travis CI) to enable CI (continuous integration)/CD (continuous deployment). By triggering the webhook provided by VCS, the microservice can be deployed in the container for collaboration with the original monolithic system.

Reconstructing the system using the Strangler Fig pattern allows the monolithic system to operate simultaneously with the Strangler application without affecting client-side users. After completing the development and testing of a Strangler service, code fragments corresponding to the functions that have been implemented by services can be removed. The migration process is implemented incrementally until all services have been migrated.

4. CASE STUDIES

This section describes the details of the architecture migration for two monolithic systems: Green Button and BBDP (Bridge between Doctors and Patients), by using the proposed MMSD approach.

4.1 Green Button

The U.S. government launched the "Green Button" [28] project in 2012. The use of Smart Grid technology lets the users get detailed information on energy consumption (mainly electrical power data) by clicking a button on the website. The users are also able to authorize their data to third-parties for diagnosing energy consumption, assisting the energy management, or cutting out unnecessary power usage to save energy fees. The "Green Button" project also released an open-source project for the Green Button system. The system provides two core features: (1) Download My Data (DMD): a user can download personal energy data hosted by the data custodian, and (2) Connect My Data (CMD): a third-party application can also obtain authorized user data from the data custodian to provide value-added services. This open-source project is divided into three sub-projects: DataCustodian, ThirdParty, and their shared Common projects. These projects were developed by the Spring framework and JSP (Java Server Pages).

In the past, we tried to develop a new third-party application to obtain user energy data from DataCustodian through CMD for data analysis and visualization. However, HTTP 404 and data rollback errors occur occasionally. Therefore, we decided to redesign

the DataCustodian project's architecture and migrate it to MSA based on the proposed MMSD approach.

4.2.1 Domain-driven design for DataCustodian of green button

We first extracted all of the nouns from the requirement specifications of Green Button, assessed whether they belong to vital classes, and used them to create an abstract domain model (See Fig. 2) by which to identify associations among classes.

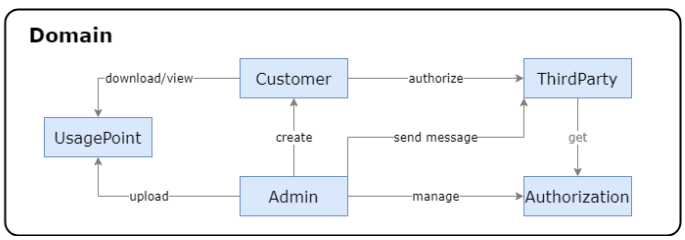


Fig. 2. Abstract domain model of DataCustodian.

We implemented the strategic design by dividing the system into four subdomains in accordance with the business capabilities. Each subdomain was mapped to a microservice (Bounded Context) with a corresponding domain model, as shown in Fig. 3. We then implemented the tactical design to derive model details.

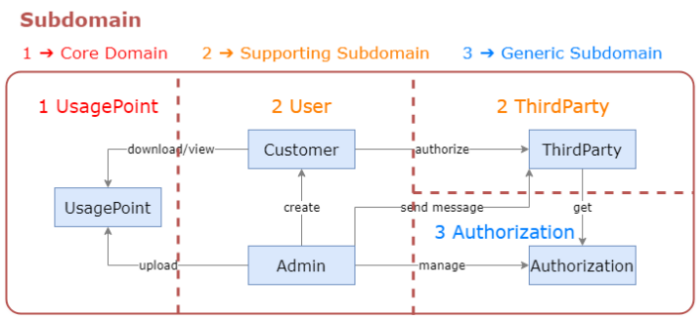


Fig. 3. Strategic design of DataCustodian.

Among the identified services, UsagePoint involves the processing of large quantities of energy data on which are performed read and write operations. Here, the Command Query Responsibility Segregation (CQRS) pattern [29] was used to design the UsagePoint service. Read and write operations were respectively separated into the DownloadMyData (DMD) service and UploadData service. The former is a dedicated query service that allows users to query energy data, whereas the latter is a dedicated command service that uploads/updates energy data. In terms of database design, both services possess the same data structure. When the UpdateData service alters the database, a data update event is published to notify the DownloadMyData service in order to maintain data consistency.

4.2.1 Service prioritization for DataCustodian in green button

The above analysis results were used to guide the splitting of DataCustodian into five microservices: User, ThirdParty, Authorization, DMD, and UploadData. The proposed service value scoring guidelines were used to select the most valuable service as the primary target of service migration. As shown in Table 2, DownloadMyData (DMD) was identified as the core service with the most profound effect on system performance, and was therefore adopted as the first Strangler service. Note that in the original system, DMD consumes considerable resources (CPU and memory). Please refer to [30] to see details of the tactical design, system mapping between the original and new system designs, and the deployment process for the migrated microservice (DMD, Download My Data).

Table 2. Service scores for application of DataCustodian to green button.

Service/Criteria	FM	PP	CF	RS	SS	Total
User	0	0	15	0	5	20
ThirdParty	0	0	10	0	5	15
Authorization	0	0	20	0	5	25
DMD	**0**	**20**	**20**	**0**	**3**	**43**
UploadData	0	20	5	0	3	28

4.2 BBDP

Bridge between Doctors and Patients (BBDP) is a mobile application developed by our lab for personal health tracking, conducting questionnaires, and assessing medical records for in-depth diagnosis and treatment. BBDP is meant to promote communication among patients, doctors, and the patients' families to mediate medical information inequity and improve the quality of medical care.

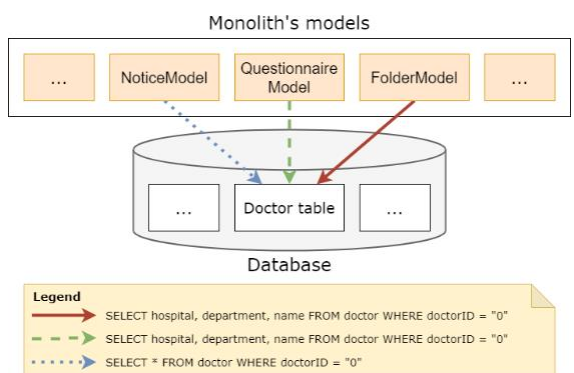


Fig. 4. Problems in existing BBDP system.

BBDP is a typical monolithic application, in which all modules affect all data items in the database. In other words, individual modules do not have clear responsibilities, and many operations are implemented multiple times in different modules. Fig. 4 illustrates the problem.

We decided to redesign the architecture and migrate it to MSA using redefined APIs with clear functions. We also sought to enhance the performance of the folder module, which allows a patient to upload image files to the doctor.

4.2.1 Domain-driven design for BBDP

We first extracted all nouns from the specifications of BBDP, assessed whether they belong to vital classes, and used them to create an abstract domain model by which to identify associations among classes (See Fig. 2).

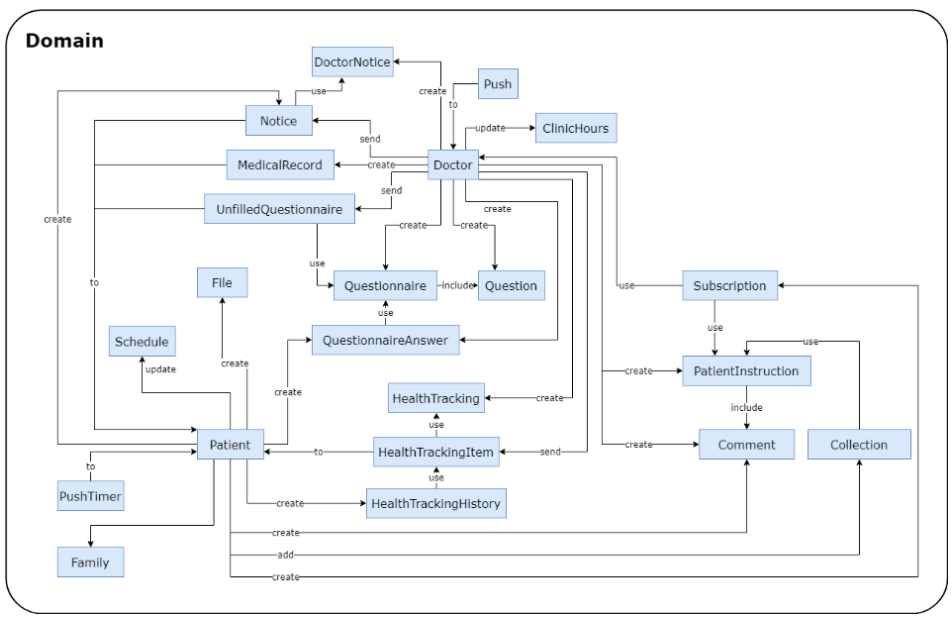


Fig. 5 Abstract model for BBDP.

We then implemented the strategic design by dividing the system into 14 subdomains in accordance with the business capabilities, including Doctor, Patient, Family, Push, PushTimer, ClinicHours, Questionnaire, Folder, HealthTracking, Notice, DoctorNotice, MedicalRecord, Schedule, and PatientInstruction. Each subdomain was mapped to a microservice (Bounded Context) with a corresponding domain model, as shown in Fig. 6. Note that core domains are marked in red, supporting subdomains are marked in orange, and generic subdomains are marked in blue. We then used "context mapping" to describe interactions among Bounded Contexts (as shown in Fig. 7). Besides, we used upstream U and downstream D to express dependencies.

The results of the strategic design were used to guide the implementation of the tactical design aimed at deriving model details, as shown in Fig. 8 (please refer to https://shorturl.at/nHMRY to browse the version of a larger image). We then mapped the modules and database tables in the original BBDP system to the identified services. We discovered that two of the modules could not be mapped to an appropriate microservice: (1) The BBDPBase64 module, which is responsible for Base64 encryption and decryption and is

not included in any business function. We decided to wrap it as a system library to enable its use by multiple services; and (2) The Doctor/Patient Suggestion module, for which the original specifications lack requirement statements. We decided to implement a new service to deal with it, such that the total number of identified services was increased to 15.

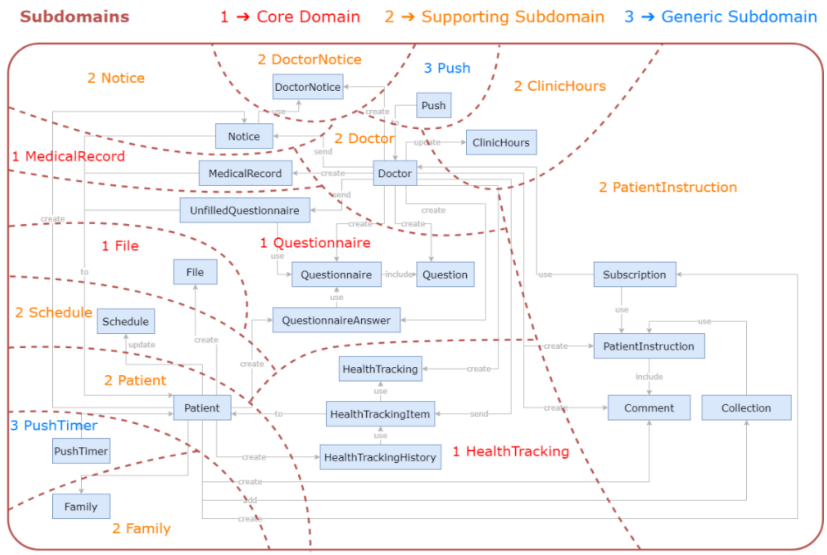


Fig. 6. Strategic design of BBDP: Subdomain design.

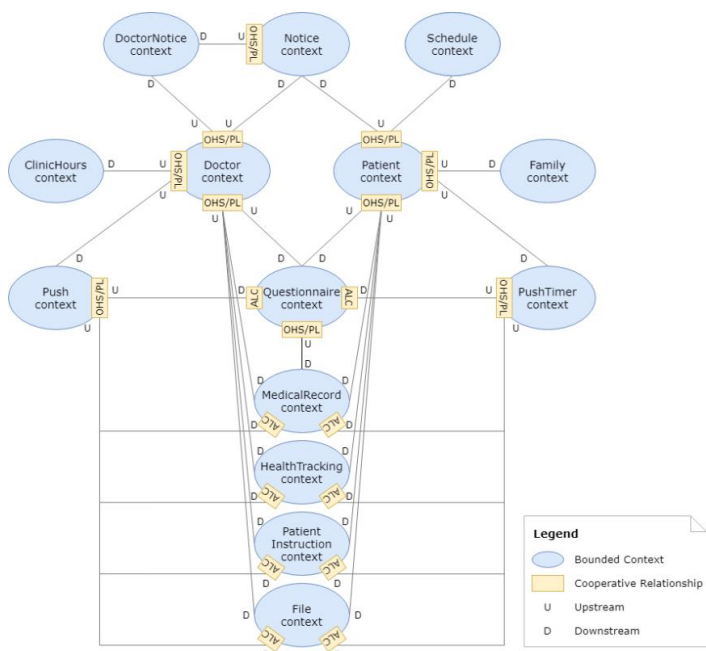


Fig. 7. Strategic design of BBDP: Subdomain design and context mapping.

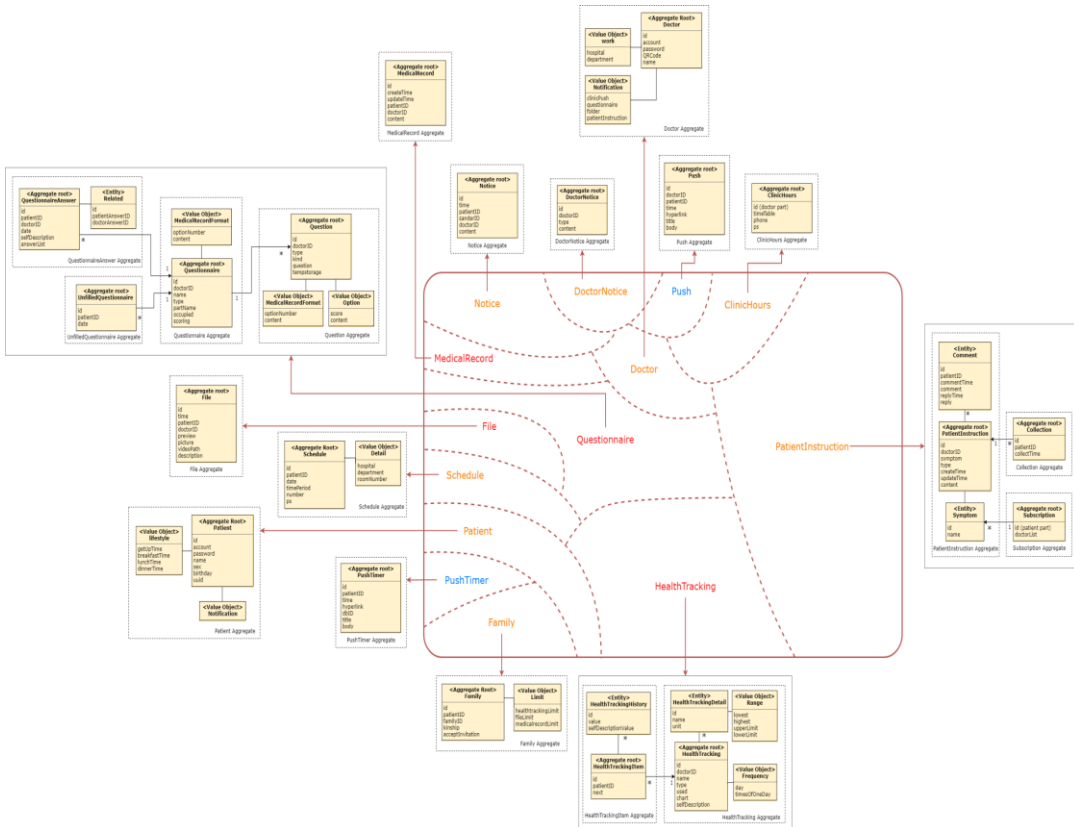


Fig. 8. Tactical design for BBDP.

4.2.1 Service prioritization and construction for BBDP

Following the completion of DDD, we used the proposed service score calculation method to prioritize services for migration. The first calculations are listed in Table 3. Note that the Folder service was adopted as the first Strangler service due to its performance issues. The second calculations are presented in Table 4. Questionnaire was adopted as the second Strangler service, due to its core functionality. The third calculations are presented in Table 5. MedicalRecord was adopted as the third Strangler service, due to its core functionality, which is strongly linked to the Questionnaire service but distinct from the reminder services.

We created new microservice projects for the Folder, Questionnaire, and MedicalRecord services. The microservices were deployed using Docker and registered using Eureka, as shown in Fig. 9. All service invocations were transmitted via an API gateway. Collaboration between migrated microservices and the monolithic system was based on REST APIs. Postman[1] test results verified that requests for the three migrated microservices produced the expected responses.

[1] https://www.postman.com/

Table 3. Service score calculations for BBDP-1.

Service/Criteria	FM	PP	CF	RS	SS	Total
Patient	0	5	5	0	0	10
Doctor	0	5	5	0	0	10
Family	0	0	5	0	0	5
Questionnaire	0	0	20	0	8	28
MedicalRecord	0	0	18	0	8	26
Folder	**0**	**20**	**15**	**0**	**10**	**45**
HealthTracking	0	0	15	0	10	25
DoctorNotice	0	0	10	0	8	18
Notice	0	0	10	0	8	18
ClinicHours	0	0	5	0	10	15
Schedule	0	0	5	0	10	15
PatientInstruction	0	0	5	0	8	13
PushTimer	0	10	10	0	0	20
Push	0	5	10	0	0	15
SystemAdmin	0	0	0	0	10	10

Table 4. Service score calculations for BBDP-2 (partial).

Service/Criteria	FM	PP	CF	RS	SS	Total
Patient	0	5	5	1	0	11
Doctor	0	5	5	1	0	11
Family	0	0	5	2	0	7
Questionnaire	**0**	**0**	**20**	**0**	**8**	**28**
MedicalRecord	0	0	18	0	8	26

Table 5. Service score calculations for BBDP-3 (partial).

Service/Criteria	FM	PP	CF	RS	SS	Total
Patient	0	5	5	2	0	12
Doctor	0	5	5	2	0	12
Family	0	0	5	4	0	9
Questionnaire	/	/	/	/	/	/
MedicalRecord	**0**	**0**	**18**	**5**	**10**	**33**

Fig. 9. Registration of BBDP services in Eureka.

5. EXPERIMENTAL EVALUATIONS

To demonstrate the feasibility and benefits of the proposed microservice migration approach, we designed and conducted quantitative experiments for the above two cases, the Green Button system and the BBDP system. Because that the modularity and main-

tainability are difficult to measure objectively, and the migrated microservices are obviously more modular and maintainable than monolithic systems, we planned to evaluate MMSD from the viewpoint of QoS (Quality of Service) and answer the following research questions:

RQ-1: Does the migrated microservices enhance reliability?

RQ-2: Does the migrated microservices improve the performance, such as response time or throughput?

RQ-3: Can any benefits be gained for the unmigrated modules in a monolithic system?

In the following sub-sections, we describe the experiment setup first and discuss the design and evaluation results for the two experiments.

5.1 Experiment Setup

As mentioned above, DMD and Folder were selected as the first Strangler services, due to the fact that they imposed performance bottlenecks in the original systems. We expected that the performance of modules in the original application could be improved through service migration. To monitor the quality of the migrated microservices, we conducted two experiments involving quantitative evaluations of Green Button and BBDP using Apache JMeter.[2] Note that Apache JMeter is an open-source Java software package commonly used to test the performance of applications deployed on servers.

The experiments were run on a 64-bit Ubuntu server (version 16.04) with an Intel 4-Core i5-6400 @ 2.7GHz with 24GB of RAM and a 1TB hard disk. All systems and services were deployed in different docker containers. We also configured the CPU to use three cores and 6GB of memory for the sake of uniformity.

5.2 Experiment 1: Green Button Migration

The first experiment on the Green Button system was used to evaluate the performance of services before and after migration, including DMD services (see Table 5) and ThirdParty services (see Table 6). The DMD modules with the highest resource consumption were migrated as microservices. We did not observe a significant improvement in the computational performance of DMD (63.2 versus 62.6 seconds); however, we observed a notable improvement in reliability, as evidenced by a reduction in the error rate (from 1.6% to 0%). We observed a significant improvement in the computational performance of the unmigrated module, ThirdParty, as evidenced by a reduction in response time from 0.73 seconds to 0.01 seconds, when migrated microservices were called at the same time. The bar charts for Tables 6 and 7 are shown in Fig. 10.

Table 6. DMD service testing before and after its migration.

AUT: *DownloadMyData (DMD)*	Before migration	After migration
Samples	1000 (threads)	
Average Response Time (ms)	63,239	62,561
Error (%)	1.6	0
Throughput (/sec)	3.1	3.1
Average response size (byte)	2,622,433	2,624,419

[2] https://jmeter.apache.org/

Table 7. ThirdParty service testing before and after the migration for DMD service.

AUT: *ThirdParty*	Before migration	After migration
Samples	1000 (threads)	
Average Response Time (ms)	734	10
Error (%)	0	0
Throughput (/sec)	3.6	3.7
Average response size (byte)	5,156	5,156

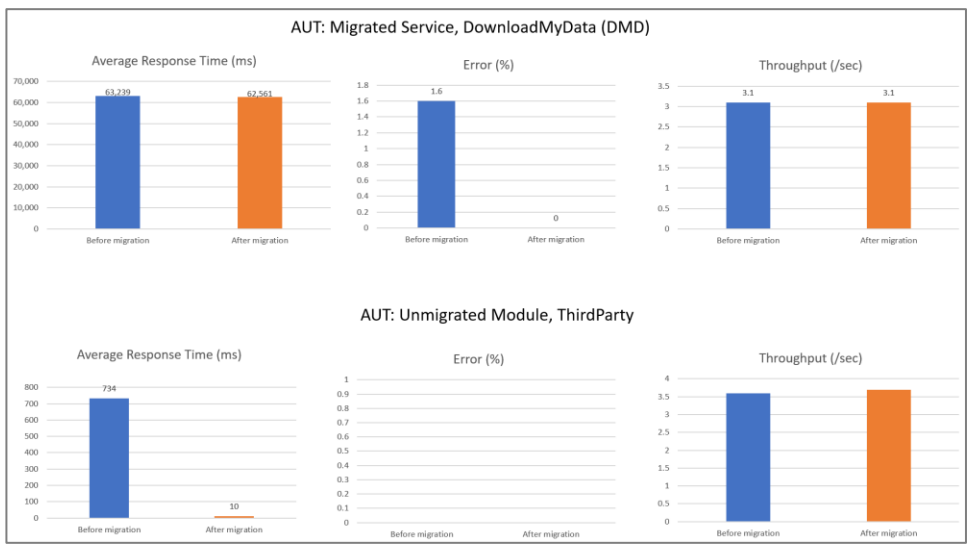


Fig. 10. Comparison for the migrated service and the unmigrated module in GreenButton.

5.3 Experiment 2: BBDP Migration

The second experiment on the BBDP system was used to evaluate the performance of services before and after migration, including the Folder service (see Table 7) and HealthTracking (see Table 8). Again, the DMD modules with the highest resource consumption were migrated as microservices. We observed a slight decrease in the computational performance of Folder services (13.67 versus 15.56 seconds); however, the performance of the unmigrated module, HealthTracking, improved significantly, as evidenced by a reduction in the response time from 3.56 seconds to 0.08 seconds when migrated microservices were called at the same time. The bar charts for Tables 8 and 9 are shown in Fig. 11.

Table 8. Folder service testing before and after its migration.

AUT: *Folder*	Before migration	After migration
Samples	1000 (threads)	
Average Response Time (ms)	13,665	15,559
Error (%)	0	0
Throughput (/sec)	10.8	10.7
Average response size (byte)	10,979,214	10,979,228

Table 9. HealthTracking service testing before and after the migration for folder service.

AUT: *HealthTracking*	Before migration	After migration
Samples	1000 (threads)	
Average Response Time (ms)	3,561	78
Error (%)	0	0
Throughput (/sec)	10.9	10.8
Average response size (byte)	282	282

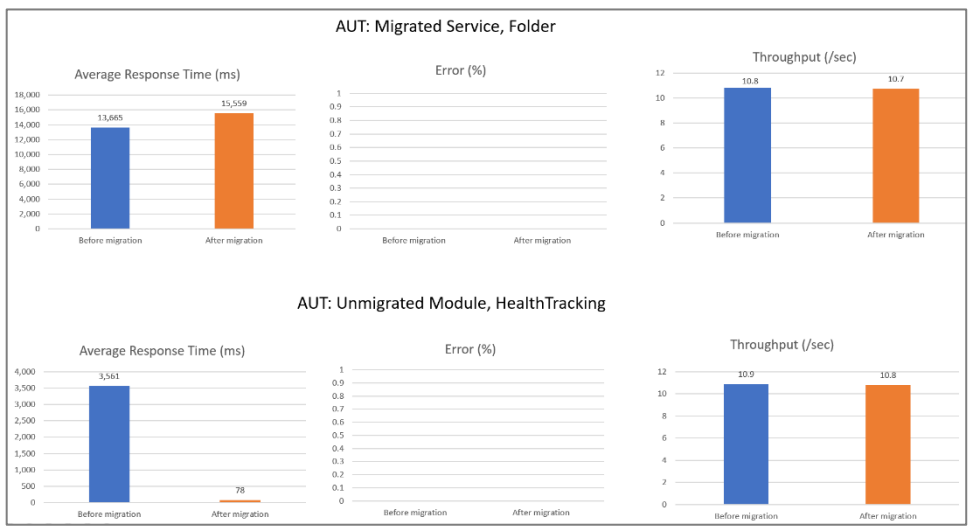


Fig. 11. Comparison for the migrated service and the unmigrated module in BBDP.

5.4 Discussion

Experiment results for GreenButton and BBDP revealed that system performance could be improved by eliminating performance bottlenecks via migration to microservices. We made two important observations that can answer the research questions:

(1) The migration of microservices can enhance reliability without compromising computational performance. Therefore, the answer to RQ-1 is yes; the number of errors for the migrated services is reduced. The answer to RQ-2 is no; microservice architecture can only keep the same level for response time and throughput.
(2) The performance of unmigrated modules in monolithic systems can be improved by migrating problematic services in order to eliminate bottlenecks. Therefore, the answer to RQ-3 is the performance of the unmigrated modules can be effectively improved.

Based on the analysis of case studies, we also conclude two possible issues/difficulties when applying MMSD: (1) preparing the environment of hosting migrated microservices is non-trivial. We need to deploy the service discovery system, API gateway, service monitor system, distributed tracing system, and even service orchestration platform (such as Kubernetes). The preliminary work is laborious for some organizations; and (2) designing the interface of a migrated microservice is not simple and even challenging. We

need to carefully ensure the correct integration of the newly-built microservice and the remainder monolith. However, since the process of applying the Strangler-Fig pattern is reversible, we can abandon a microservice if the integrations testing fails or some unexpected errors occur when using migrated services.

6. CONCLUSIONS

This paper reports on the application of the Strangler Fig pattern in software migration from a monolithic architecture to MSA. The proposed scheme involves the incremental extraction and construction of microservices based on DDD (Domain-Driven Design). The Bounded Context in DDD makes it possible to assign specific responsibilities to each service and clearly define the scope. The identified services are mapped to the models and database of the original system to facilitate service implementation. The proposed scheme proved effective in identifying and establishing appropriate microservices when applied to two existing monolithic applications, Green Button and BBDP. Architecture migration was shown to enhance the reliability of migrated microservices and the efficiency of unmigrated modules.

In the future, we will apply the proposed MMSD approach to additional monolithic systems, conduct scalability experiments in the cloud, and apply the sidecar pattern to enhance service monitoring, circuit breaker, and load balancing features.

REFERENCES

1. B. Foote and J. Yoder, "Big ball of mud," *Pattern Languages of Program Design*, Vol. 4, 1997, pp. 654-692.
2. G. Kousiouris, S. Tsarsitalidis, E. Psomakelis, *et al.*, "A microservice-based framework for integrating IoT management platforms, semantic and AI services for supply chain management," *ICT Express*, Vol. 5, 2019, pp. 141-145.
3. C. Richardson, *Microservices Patterns: With Examples in Java*, Simon and Schuster, USA, 2018.
4. S.-P. Ma, C.-Y. Fan, Y. Chuang, I. H. Liu, and C.-W. Lan, "Graph-based and scenario-driven microservice analysis, retrieval, and testing," *Future Generation Computer Systems*, Vol. 100, 2019, pp. 724-735.
5. C. Y. Fan and S. P. Ma, "Migrating monolithic mobile application to microservice architecture: An experiment report," in *Proceedings of IEEE International Conference on AI & Mobile Services*, 2017, pp. 109-112.
6. M. Fowler, "StranglerFigApplication," https://martinfowler.com/bliki/StranglerFigApplication.html, 2004
7. M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Proceedings of European Conference on Service-Oriented and Cloud Computing*, 2016, pp. 185-200.
8. M. J. Amiri, "Object-aware identification of microservices," in *Proceedings of IEEE International Conference on Services Computing*, 2018, pp. 253-256.
9. L. Baresi, M. Garriga, and A. de Renzis, "Microservices identification through interface analysis," in *Proceedings of European Conference on Service-Oriented and*

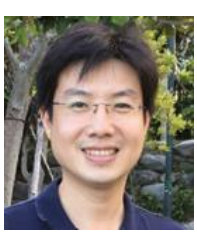
Cloud Computing, 2017, pp. 19-33.
10. R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference*, 2017, pp. 466-475.
11. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, USA, 2004.
12. E. Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries*, Dog Ear Publishing, USA, 2014.
13. R. H. Steinegger, P. Giessler, B. Hippchen, and S. Abeck, "Overview of a domain-driven design approach to build microservice-based applications," in *Proceedings of the 3rd International Conference on Advances and Trends in Software Engineering*, 2017, pp. 79-87.
14. B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, "Designing microservice-based applications by using a domain-driven design approach," *International Journal on Advances in Software*, Vol. 10, 2017, p. 2017.
15. F. Rademacher, J. Sorgalla, and S. Sachweh, "Challenges of domain-driven microservice design: a model-driven perspective," *IEEE Software*, Vol. 35, 2018, pp. 36-43.
16. M. Rizki, A. Fajar, and A. Retnowardhani, "Designing online healthcare using DDD in microservices architecture," *Journal of Physics: Conference Series*, Vol. 1898, 2021, pp. 1-5.
17. I. J. Munezero, D.-T. Mukasa, B. Kanagwa, and J. Balikuddembe, "Partitioning microservices: A domain engineering approach," in *Proceedings of IEEE/ACM Symposium on Software Engineering in Africa*, 2018, pp. 43-49.
18. H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Software*, Vol. 35, 2018, pp. 44-49.
19. S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, O'Reilly Media, USA, 2019.
20. R. Petrasch, "Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication," in *Proceedings of the 14th International Joint Conference on Computer Science and Software Engineering*, 2017, pp. 1-4.
21. R. A. Schmidt and M. Thiry, "Microservices identification strategies : A review focused on model-driven engineering and domain driven design approaches," in *Proceedings of the 15th Iberian Conference on Information Systems and Technologies*. 2020, pp. 1-6.
22. F. Montesi and J. Weber, "Circuit breakers, discovery, and API gateways in microservices," *arXiv Preprint*, 2016, arXiv:1609.05830.
23. C. Richardson, *Microservices Patterns: With Examples in Java*, 2019, Manning Publications, Switzerland, Europe.
24. P. S. Kocher, *Microservices and Containers*, Addison-Wesley Professional, USA, 2018.
25. D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, Vol. 4, 2017, pp. 22-32.
26. A. Cowart, "The phases of a microservices project," https://headspring.com/2018/02/05/microservices-project-phases/, 2018.

27. A. Deshpande and N. P. Singh, "Challenges and patterns for modernizing a monolithic application into microservices," https://developer.ibm.com/articles/challenges-and-patterns-for-modernizing-a-monolithic-application-into-microservices/, 2020.
28. D. S. Sayogo and T. A. Pardo, "Understanding smart data disclosure policy success: the case of Green Button," in *Proceedings of the 14th ACM Annual International Conference on Digital Government Research*, 2013, pp. 72-81.
29. C. Richardson, "Pattern: Command query responsibility segregation," https://microservices.io/patterns/data/cqrs.html, 2020.
30. C. Y. Li, S. P. Ma, and T. W. Lu, "Microservice migration using strangler fig pattern: A case study on the green button system," in *Proceedings of International Computer Symposium*, 2020, pp. 519-524.

Shang-Pin Ma (馬尚彬) received his Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in 2007. Dr. Ma is currently a Professor in the Department of Computer Science and Engineering at National Taiwan Ocean University. His research interests include service-oriented computing, software engineering, mobile computing, and chatbot architecture.

Chia-Yu Li (李佳育) received her BS (2018) and MS (2020) degrees from the Department of Computer Science and Engineering, National Taiwan Ocean University, Taiwan. Her research interests include software engineering and microservice architecture.

Wen-Tin Lee (李文廷) received his Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in 2008. Dr. Lee is the Department Head and an Associative Professor in the Department of Software Engineering and Management at National Kaohsiung Normal University. His research interests include software engineering, service-oriented computing, and deep learning.



Shin-Jie Lee (李信杰) is an Associate Professor in Computer and Network Center at National Cheng Kung University in Taiwan and holds joint appointments from Department of Computer Science and Information Engineering at NCKU. His current research interests include software engineering and service-oriented computing. He received his Ph.D. degree in Computer Science and Information Engineering from National Central University in Taiwan in 2007.