Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example

Chapter · January 2020

DOI: 10.1007/978-3-030-33624-0_4

3 authors, including:

Alan Megargel
Singapore Management University
7 PUBLICATIONS **63** CITATIONS
SEE PROFILE

Venky Shankararaman
Singapore Management University
91 PUBLICATIONS **770** CITATIONS
SEE PROFILE

1-2020

Migrating from monoliths to cloud-based microservices: A banking industry example

Alan MEGARGEL
Singapore Management University, ALANMEGARGEL@SMU.EDU.SG

Venky SHANKARARAMAN
Singapore Management University, venks@smu.edu.sg

David K. WALKER

Chapter 1:
Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example

Alan Megargel, Venky Shankararaman and David K. Walker

School of Information Systems, Singapore Management University

Abstract: As more organizations are placing cloud computing at the heart of their digital transformation strategy, it is important that they adopt appropriate architectures and development methodologies to leverage the full benefits of the cloud. A mere "lift and move" approach, where traditional monolith applications are moved to the cloud will not support the demands of digital services. While, monolithic applications may be easier to develop and control, they are inflexible to change and lack the scalability needed for cloud environments. Microservices architecture, which adopts some of the concepts and principles from service-oriented architecture, provides a number of benefits when developing an enterprise application as compared to a monolithic architecture. Microservices architecture offers agility and faster development and deployment cycles, scalability of selected functionality, and the ability to develop solutions using a mixture of technologies. Microservices architecture aims to decompose a monolithic application into a set of independent services which communicate with each other through open APIs or highly scalable messaging. In short, microservices architecture is more suited for building agile and scalable cloud-based solutions. This chapter provides a practice-based view and comparison between the monolithic and microservices styles of application architecture in the context of cloud computing, and proposes a methodology for transitioning from monoliths to cloud-based microservices.

Keywords: Microservces Architecture, Monolithic Architecture, Cloud-Based, Microservice Identification, Migration from Monolith to Microservices

1.1 Introduction

Digital transformation requires organizations to be nimble and adopt accelerated innovation methods which enable the delivery of new digital services to customers, partners and employees. To achieve this, organizations are looking towards building flexible cloud-based applications, whereby it is easier to add and update digital services as requirements and technologies change. Legacy monolithic applications might be operationally acceptable on a day-to-day basis but these applications are not well suited for building digital services. Traditional monolithic architecture and software development methods remain a stumbling block for driving digital transformation. In order to efficiently drive digital transformation, organizations are

exploring a new software development methodology and architecture, "cloud-based microservices architecture", whereby IT solutions can be organized around granular business capabilities which can be rapidly assembled to create new cloud-based digital experience applications. The "microservices" architecture is a style and method of developing software applications more quickly by building them as collections of independent, small, modular services. Organizations are currently faced with two challenges, namely; how to build new applications using a microservices architecture, and how to migrate from a monolith to a cloud-based microservices architecture. This chapter provides practical guidance and a methodical approach to address these two challenges.

A single monolith is typically composed of tens or hundreds of business functions, which are deployed together in one software release. Microservices on the other hand typically encapsulate a single business function which can be scaled separately and deployed separately. It is possible to develop a large enterprise application for cloud deployment by assembling and orchestrating a set of microservices, as an alternative to developing a monolith.

In this chapter, we first discuss the challenges of monolithic applications in terms of technology stack, scalability, change management, and deployment. We then propose a microservices architecture, as an alternative, and provide a comparison with the monolith. We then propose a methodical approach to transitioning from a monolith application to a cloud-based microservices application, both from the perspective of building new solutions from scratch and migrating existing solutions built as monoliths. Finally, we conclude with a summary and ideas for future work.

1.2 Monolithic Applications: Background and Challenges

A monolithic application, or "Monolith", describes a legacy style of application architecture which does not consider modularity as a design principle. Originally, the term "monolithic" was used to describe large mainframe applications [23], which are self-contained and become increasingly complex to maintain as the number of functions they support increases over many years of version updates. Following the mainframe era, incarnations of the monolithic architecture style emerged, namely; client-server architecture, and three-tier web application architecture [22]. A common characteristic of all forms of monolithic application is the presence of three distinct architecture layers; the user interface layer, the business logic layer, and the database layer. A simplified illustration of these three layers, or tiers, is provided in Figure 1 below.

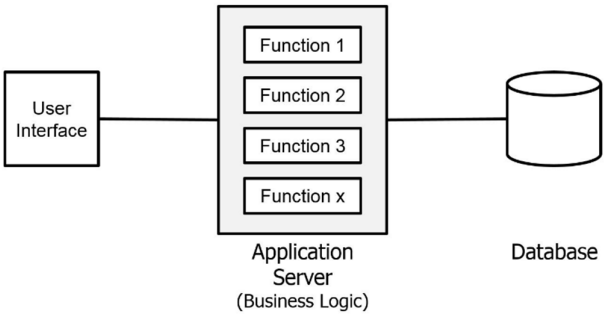

Figure 1 Monolithic Application (or "Monolith")

The defining characteristic of a monolith is that all of the business logic is developed and deployed together onto the middle tier, typically hosted on an application server. More broadly, beyond mainframes, a monolith can be described as any block of code that includes multiple functions. Business logic is coded into functions, each fulfilling a specific business capability, e.g.; order management, or account maintenance. The first release of an application might include several tens of functions, and with subsequent releases, the application might grow to include several hundreds of functions [20]. Before discussing their issues it is only fair to state that monoliths, especially mainframe systems, are highly performing in terms of response time and throughput, and are highly resilient and reliable [15]. Many established banks are still relying on 1970's mainframe technology for their core banking systems [15]. However, monoliths are not suitable for cloud deployment due to several reasons which are explained as follows:

Technology Stack

In a monolith, the functions which implement business logic are all typically written using the same programming language as was popular and relevant at the time the original application was developed. Mainframe applications, for example, are written using the COBOL language, and any extensions or subsequent version releases of the application must also be written in COBOL. Developers are locked-in to the original technology stack, and as such are not free to develop new functions using modern application frameworks or languages suitable for cloud deployment [22]. As the number of functions increases, a monolith becomes more complex, and requires a larger team of developers to support the application [20].

The functions implemented in a monolith, all developed using the same programming language, must interact with each other using native method calls, and are therefore tightly-coupled [20]. For cloud deployment, loosely-coupled functions are more suitable [8]. Loose-coupling of functions within a monolith is not possible, for example, it is not possible for function-A to interact with function-B using native method calls if the two functions are deployed onto different servers.

Scalability

Functions within a monolith collectively share the resources (CPU and memory) available on the host system. The amount of system recourses consumed by each function varies depending on demand. A high demand function, for example one that has a high number of requests via the user interface, or one that is computationally intensive, may at one point consume all of the available resources on the host machine. Therefore, scalability within a single monolith is limited [20].

Vertically scaling the monolith by increasing the system memory would be an option, but a high demand function would eventually consume the additional memory as well. Since functions within a monolith are tightly-coupled and cannot be individually deployed in separate systems, as mentioned in the previous section, the best and most widely used option would be to scale the monolith horizontally.

Horizontal scaling of a monolith, as illustrated in Figure 2 below, involves adding whole new redundant servers [4], as many as necessary, in order to handle any number of incoming requests through the user interface. A load balancer is needed in order to split the load of incoming requests evenly between the servers. Session replication between servers is needed so that a single user session can span across servers, or alternatively "sticky" session can be configured to ensure that all requests from the same user are routed consistently to the same server. Either way, database replication is needed in order to ensure that all redundant instances of the database are kept current. Horizontal scaling adds cost and complexity to a monolith, making it impractical for cloud deployment.

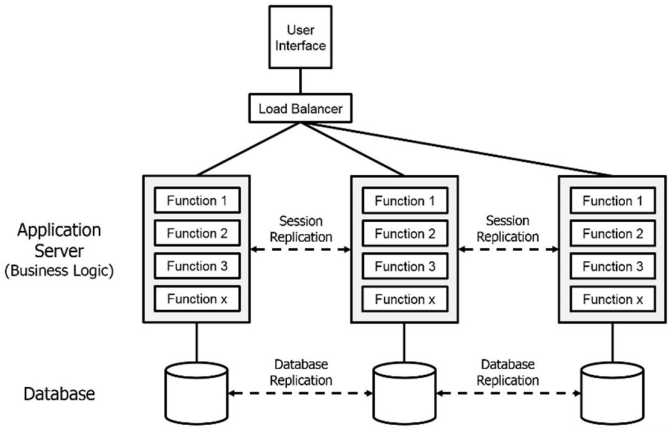

Figure 2 Horizontal Scaling of a Monolith

Change Management

As mentioned above, a monolith becomes more complex as the number of functions increases over time, requiring a larger support team. Functions which interact using native method calls are tightly-coupled and interdependent, and therefore are susceptible to change. A change to one function might impact any other function which interacts with that function. Due to these interdependencies, testing only the function which has changed would be insufficient, rather the entire application should be retested to ensure there is no impact due to the change. Retesting an entire application implies that all test cases need to be regression tested, ensuring that tests which are expected to pass still pass, and tests which are expected to fail still fail.

Because a change to any function requires the entire application to be retested, change management processes for monoliths are complex. Test cases need to be maintained. Regression tests need to be planned and scheduled. Test results need to be reviewed. Any test failures cause the entire application to revert back to the development team for bug fixing. Monoliths are typically managed using a waterfall software development lifecycle (SDLC) methodology [17], which requires the entire application to be promoted through a sequence of states, namely; development, system integration testing (SIT), user acceptance testing (UAT), and production. Typically large enterprises such as banks have specialized change management teams who plan, schedule and execute changes. Incident management teams report that 80 to 90 percent of production problems occur due to improperly tested changes as the root cause [15], even with rigorous testing practices in place. Due to the risk of production problems, banks may schedule the re-deployment of monoliths to occur only once per month, even for routine enhancements. The careful and rigorous testing practices implemented for monoliths can inhibit the time-to-market of new customer experience driven innovations.

Deployment

Individual functions within a monolith cannot be individually deployed, rather the entire application must be deployed. The deployment package for a monolith is typically one large file. For example, the deployment package for a java web application is a single web application resource (WAR) file. Other types of single file deployment archives include; java archive (JAR), enterprise java bean (EJB), tape archive (TAR) for Linux/Unix, and dynamic link library (DLL) for Windows. The deployment archive for a monolith increases in size as the number of functions within the monolith increases.

Individual functions within a monolith cannot be individually restarted after deployment, rather the entire application must be restarted. The implication of this is that the entire application would be unavailable to users while it is being restarted, unless the application is deployed in a high availability (HA) configuration of servers, in which case the application could be deployed and restarted on one HA

server at a time. Large monoliths, with hundreds of functions, can take 20 to 40 minutes to restart [20].

The size of the deployment archive, together with long restart times, plus the fact that the entire application must be re-deployed each time there is a change, makes the use of modern DevOps methods and tools challenging if not impractical for large monoliths. This would be the case for cloud deployments as well as for on-premises deployments of monoliths.

1.3 Microservices: A Cloud-based Alternative

Microservices are "a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services" [24]. A microservice encapsulates a function, or a business capability [5], which owns its own data [16], and can be independently deployed and independently scaled [20]. Microservices can encapsulate business entities (e.g.; Product, Customer, Account) or can encapsulate business activities which orchestrate multiple business entities (e.g.; Credit Evaluation, Trade Settlement) [1, 8].

An atomic microservice [8] is a fine-grained service which encapsulates the functionality and data of a single business entity such as Product. In this example, the Product service owns product data, i.e.; if any other service requires product data it must access it via the Product service interface. The Product service exposes functions or operations via its interface such as; GET product data, POST (create) new product data, PUT (update) existing product data, and DELETE product data. Atomic microservices represent the smallest reusable software modules which cannot usefully be further sub-divided or decomposed [8].

A composite microservice [8] is a course-grained service which encapsulates the functionality of a single business activity, such as Fund Transfer. In this example, the Fund Transfer service orchestrates an end-to-end process by invoking the operations of several atomic microservices in a sequence which fulfils the business activity, as illustrated in Figure 3 below. Composite microservices can also perform transaction management (e.g. commit or rollback) across the orchestration.

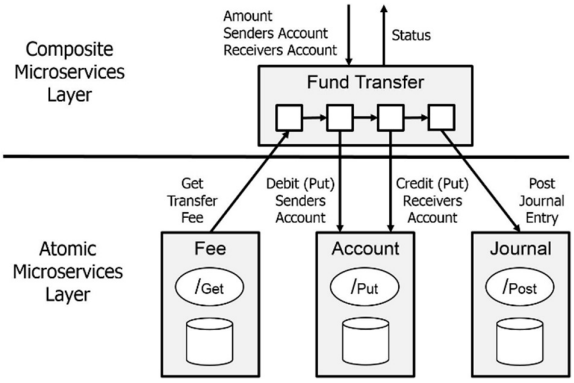

Figure 3 Microservice Layers (with Fund Transfer example)

In a microservices layered architecture, the service integration and orchestration responsibilities previously handled by an enterprise service bus (ESB) are now transferred to and dispersed among composite microservices [8]. Alternatively, the communication between microservices can be event-based [2], in which case the end-to-end process logic is distributed among the microservices. With this latter event-based approach, the end-to-end process logic can be reconstructed using service discovery and architecture recovery tools [3].

Microservices architecture (MSA) principles are similar to those established for SOA, with some additions. With regards to legacy issues associated with monoliths, the objectives of SOA and MSA are similar [2]. Both SOA and MSA aim to transform inflexible legacy architectures into services-based architectures which are more flexible and agile for developing new innovative digital solutions [8]. In complex organizations like traditional banks, SOA maturity is key to overcoming legacy systems as an inhibiter for digital banking transformation [15]. While SOA is a key enabler for the agility of on-premises solutions in the presence of monolithic legacy systems, this architecture does not translate well onto the cloud where monolith implementations are impractical. As such, MSA is now a key enabler for the agility of cloud-based solutions, provided that microservices are designed at the right level of encapsulation, or boundary context [5]. A set of MSA boundary-setting design principles are provided as follows:

MSA Boundary-Setting Design Principles

P1. Do one thing well – Microservices should be highly cohesive [5, 7] in that they encapsulate elements (methods and data) that belong together. A microservice has a specific responsibility enforced by explicit boundaries. It is the only source of a function or truth; i.e. the microservice is designed to be the single place to get, add, or change a "thing". A microservice should "do one thing well".

P2. No bigger than a squad - Each microservice is small enough that it can be built and maintained by a squad (small team) working independently [20]. A single quad / team should comfortably own a microservice, whereby the full context of the microservice is able to be understood by a single person. The microservice should ideally be less than a few hundred lines of code, or zero code using a GUI-driven designer studio. Smaller microservices are optimised to be rewritten / refactored.

P3. Grouping like data - Data and its operations set boundaries. The functional boundary of a microservice is based on the data that it owns, the operations it performs (e.g. REST resources), and the views it provides on that data [5, 7]. Data that is closely related belongs under the same microservice; e.g. data needed for a single API call often (but not always) belongs to a single microservice. If putting data together simplifies the microservice APIs, and interactions, then that is a good thing. Conversely, if separating data does not adversely impact APIs or code complexity, and does not result in a trivially small microservice, then that data might make sense to separate into two microservices.

P4. Don't share data stores – Only one microservice is to own its underlying data [5]. This implies moving away from normalized and centralised shared data stores. Microservices that need to share data, can do so via API interaction or event-based interaction.

P5. A few tables only - Typically there should only be a small number of data stores (e.g. tables) underlying a microservice; i.e. 1 to 3 tables is often the range. Data store selection for a microservice should be optimised using fit for purpose styles; e.g. in-memory data grid, relational database (SQL), or key-value pair (No-SQL).

P6. Independent technology selection – Unlike monoliths, the small size of services allows for flexibility in technology selection. Often a business requirement or constraint may dictate a specific technology choice. In other cases, technology choice may be driven by engineering skills, preference and familiarity.

P7. Independent release cadence – Microservices should be loosely coupled [7] and therefore should have their own release cadence and evolve independently. It should always be possible to deploy a microservice without redeploying any other microservices. Microservices that must always be released together could be redesigned and merged into one microservice.

P8. Limit chatty microservices – Any interdependence between atomic microservices should be removed. If two or more microservices are constantly chatty (interacting), then that's a strong indication of tight coupling [5, 7], and these microservices should be merged into one. Note: If principle P1 is followed ("do one thing well"), then there should be no chatty interdependent microservices.

Cloud Deployment of Microservices

Following the above MSA boundary-setting design principles, highly cohesive and loosely coupled microservices are more practical for cloud deployment as compared to monoliths. Microservices can be deployed independently and can be scaled independently, and are small enough in size that automated build, test, and deploy scripts can be implemented using agile DevOps methods and tools [17].

The small size of the deployment objects also makes containerization practical, using Docker or similar technology [19], whereby each microservice is deployed inside a separate virtual machine image which then can be run (instantiated) any number of times on any number of different host systems as self-contained lightweight containers which can be scaled out elastically. For example, a high demand Product microservice can be instantiated into another active-active load-balanced container during the peak load period, and then the redundant container can be later removed as the load subsides.

Cloud-based microservices are exposed to internal user interfaces and external third party applications via an API Gateway [20]. An API gateway provides a single point of entry into the microservices, as well as a single point of control. Features of an API Gateway include; a) user authentication, b) user authorization to access specific microservices, c) transformation between various data formats (e.g. JSON, XML), translation between various transport protocols (e.g. HTTP, AMQP), d) scripting for aggregating or orchestrating multiple microservices in order to reduce network traffic. Figure 4 below illustrates a microservices-based architecture.

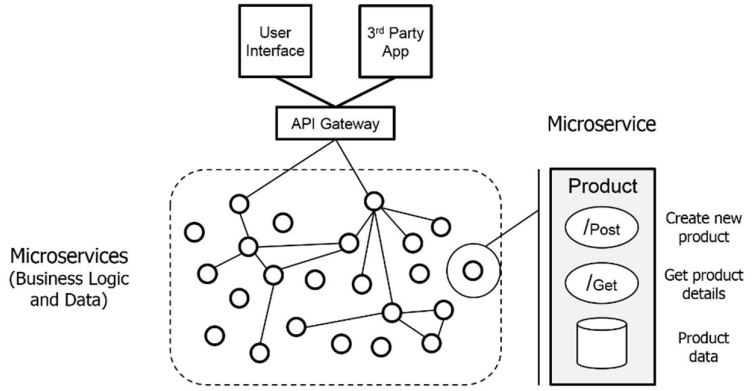

Figure 4 Microservices-based Architecture

Challenges with Microservices Deployment

The complexity of a microservices-based architecture increases over time as the number of deployed microservices increases [19, 20]. Monitoring and management tools are needed in order to; a) monitor the runtime status of microservices and

restart any which have stopped, b) monitor the loading on microservices and manage the elastic scaling of active-active load-balanced containers (instances) accordingly, and c) provide a framework for microservice discovery so that, for example, a composite microservice can locate a newly redeployed atomic microservice which gets assigned a new IP address.

Another complexity arises when interdependent microservices are located on different host systems across a wide area network (WAN); e.g. microservice 'A' requests data from microservice 'B'. In such cases, synchronous request-reply interactions would cause high network traffic across the WAN. A better approach would be to use an asynchronous event-based interaction [2] across the WAN, whereby microservice 'B' publishes data, whenever it becomes available (i.e. the event), to all microservices which have subscribed to that data. Similarly, if the same service 'A' was instantiated on multiple host systems across the WAN, there arises complexity and design challenges around how to ensure availability and/or eventual consistency [18] of data across the WAN.

Monolith vs Microserves Feature Comparison

Based on what has been discussed so far in this chapter, a features comparison between monoliths and microservices is summarized in Table 1 below.

Feature	Monolith	Microservices
Technology Stack	• Locked-in to original technology stack and framework. • All functions developed with one programming language.	• Each microservice can be developed using a different technology, fit for purpose, or based on developer preference.
Scalability	• Functions within a monolith cannot be scaled independently. • Horizontal scaling of the entire monolith is necessary.	• Each microservice can be scaled independently via containers. • Tools are needed for monitoring and managing containers.
Change Management	• For any small change, the entire monolith needs to be retested. • Change/testing processes are complex and time consuming.	• Microservices are small and can be tested quickly. • Microservices have independent release cadences.
Deployment	• Deployment file is large, slow to startup, may incur downtime. • Use of agile DevOps methods and tools is not practical.	• Microservices can be deployed independently. • Use of agile DevOps methods and tools is appropriate.

Table 1 Monolith vs Microservices Feature Comparison

1.4 Building Cloud-based Applications from Day One

Many established enterprises which are encumbered with inflexible monolithic systems are beginning to transition to a microservices architecture. Newly created enterprises have an option to build a cloud-based microservices architecture from day one, rather than to buy or build monolithic systems. In such green-field

scenarios, one of the main challenges faced by architects is the identification of candidate microservices which are highly cohesive and loosely coupled [5, 7].

Without reference to any existing monolith which can be used as a starting point for microservices decomposition, architects can take a top down approach starting with a set of business requirements, then deriving a set of business process models and/or business capability models [5], and finally decomposing those models into a set of microservice candidates. Capability-based services can be distinguished in layers as shown in Table 2 below [5]:

Service Type	Description
Business Process Service	Stateful services which orchestrate automated composite business and data services, including human interaction.
Composite Business Service	Automated services which provide business logic, by orchestrating atomic business and data services.
Composite Data Service	Automated services which provide data, by orchestrating atomic business and data services.
Atomic Business Service	Automated services which provide atomic business logic functionality; e.g. a pricing calculator.
Atomic Data Service	Automated services with provide atomic data manipulation functionality; e.g. CRUD (create, read, update, delete) product information.

Table 2 Capability-Based Service Types

Even without an existing monolith as a reference, a bottom up approach for microservices identification can be used, provided there exists a data model of the target application in the form of a unified modeling language (UML) compliant entity relationship diagram (ERD) and use cases. Service Cutter [7] is a tool which can assist architects in identifying microservice candidates which are highly cohesive and loosely coupled. Using the ERD and use cases as inputs, the Service Cutter tool extracts the "building blocks" of an application, referred to as "nanoentities", which are to be encapsulated and owned by microservices. These nanoentities are; a) **Data** which is exclusively owned and maintained/manipulated by a microservice, b) **Operations** which are the business rules/logic exclusively provided by a microservice, and c) **Artifacts** which are a "collection of data and operations results transformed into a specific format" e.g. a business report which is exclusively provided by a service [7]. Using a predefined "coupling criteria", the relationship between each pair of nanoentities in the model is scored, and finally a clustering algorithm is used to identify the candidate microservices [7].

Another source of information which can support bottom up microservices identification, in the absence of a monolith as a reference, are industry specific models. The banking industry, for example, has produced a number of widely used information models. The Banking Industry Architecture Network (BIAN) is a consortium of over 30 banks, technology vendors, and universities, which have collaborated on a service decomposition framework for banks [11, 26]. The BIAN

Service Landscape, as it is called, is a decomposition of a generic universal bank (retail, corporate, and investment banking) into a finite set of service domains which cannot be useably further decomposed. As shown in Figure 5 below, the framework is organized into three levels; a) business area, b) business domain, and c) service domain. The "Loan" service domain, for example, can encapsulate all of the business logic and data for loans. Even in the absence of a data model, this framework can be a good starting point for architects to identify candidate microservices.

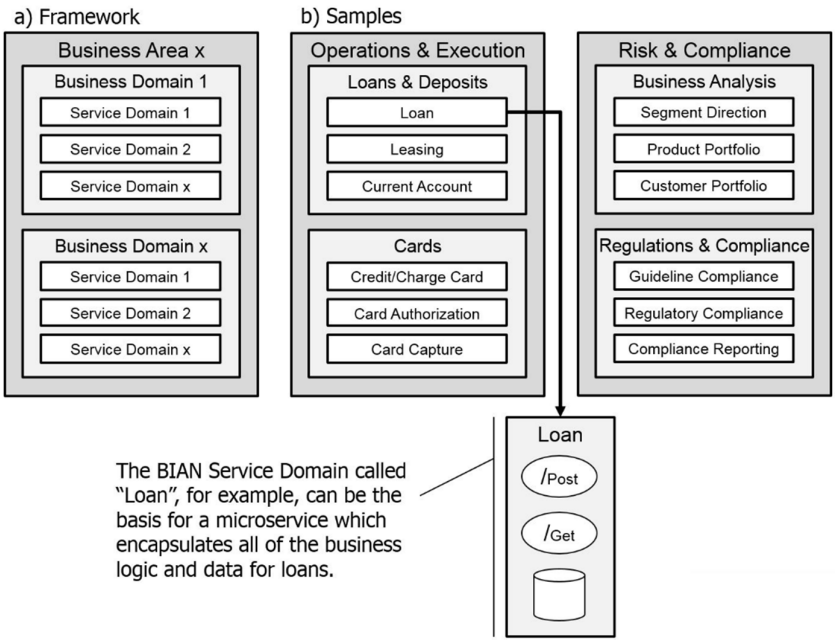

Figure 5 BIAN Service Landscape (Sample)

Technology vendor supplied data warehouse models are another source of information which can support bottom up microservices identification. The two most widely used data warehouse models in the banking industry are; Teradata Financial Services Logical Data Model (FSLDM), and IBM Information Framework (IFW) Banking Data Model. Each of these vendor supplied information models comes out-of-the-box with a set of core banking entities, also referred to as "subject areas" as illustrated in Figure 6 below. Data warehouses are organized into subject areas in order to support "subject area experts" i.e. data scientists / analysts whom are tasked to help bank management make decisions around; product, party (customer), channel, campaign, and others. The FSLDM and IFW banking industry models are improved overtime as requirements from many banks are incorporated, and therefore have become industry standards [26]. While these standard subject

areas are course-grained at a high level of abstraction, they suggest a good baseline for further decomposition into more fine-grained microservices.

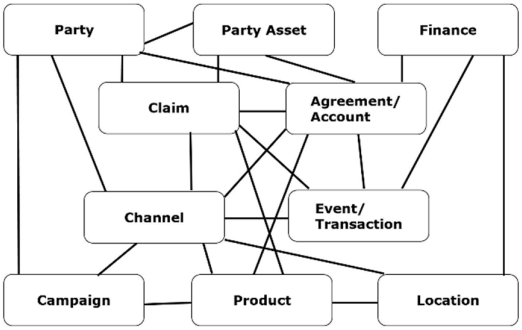

Figure 6 Teradata FSLDM Subject Areas

Each subject area has a default set of attributes which are extensible. Figure 7 below shows a worked example for the Account/Agreement subject area.

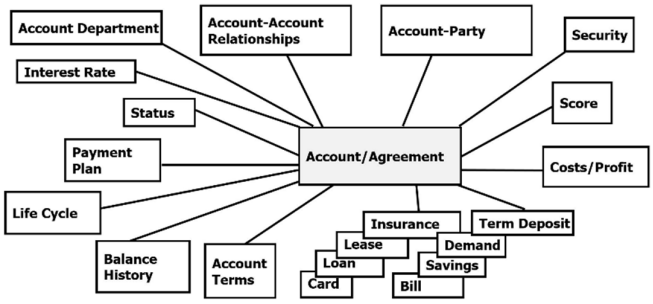

Figure 7 Attributes of FSLDM Account/Agreement Subject Area

Each subject area has an extensible set of relationships with other subject areas. Figure 8 below illustrates a worked example for the Campaign subject area. These relationship maps are useful for identifying composite services [5], as well as inter-process communications [20].

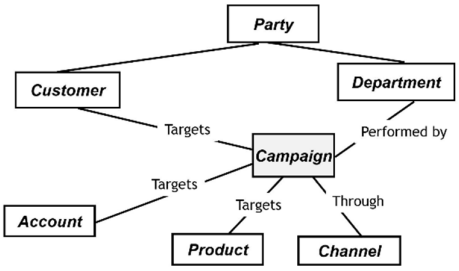

Figure 8 Relationships around FSLDM Campaign Subject Area

Each vendor supplied information model comes with a baseline ERD as shown in Figure 9 below. Data warehouse ERDs are typically implemented using a "star schema", whereby a normalized "fact" table is related to multiple de-normalized "dimension" tables. Dimension tables are useful for identifying atomic data services [5], or data nanoentities [7]. Fact tables are useful for identifying artifact nanoentities [7]. The "Customer" dimension table, for example, can encapsulate all of the business logic and data for customers.

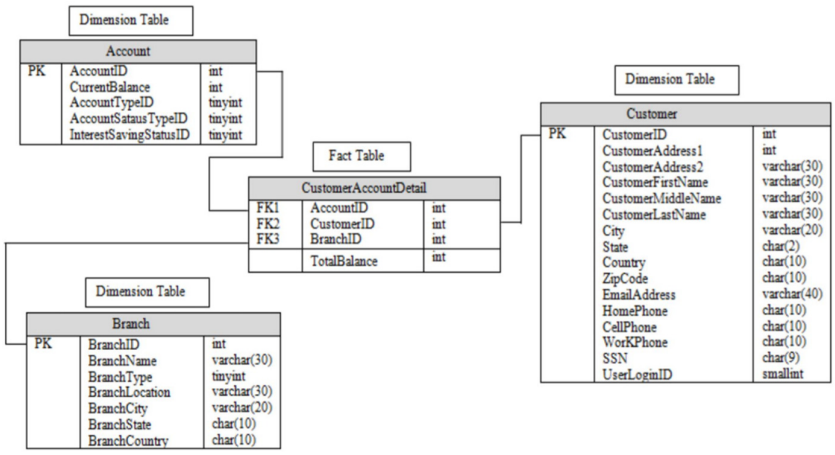

Figure 9 IBM IFW Banking Data Model (Sample)

Data warehouse models exist for other industries as well, for example healthcare; e.g. HealthCatalyst Enterprise Data Model, Oracle Healthcare Data Model. In the absence of an existing monolith to reference, it is a challenge for architects to decompose the functional boundaries of an enterprise into microservices at an optimum level of cohesiveness, and loose coupling [2]. Industry specific models offer a good starting point.

The development cycle for a cloud-based application starts with a microservice identification phase. At the end of this phase, the architect will have produced a

library of microservice interface definitions, typically in the form of a Swagger Definition File for REST-based microservices, or a Web Service Description Language (WSDL) for SOAP-based microservices [13]. Interface definitions serve as a software specification, enabling concurrent development by the backend microservice development team and the frontend user interface development team.

In a microservices-based application, each individual microservice; can be developed using a different programming language, can be developed and maintained by a small team, and can be deployed and scaled separately. A GUI-driven development tool such as TIBCO BusinessWorks Container Edition (BWCE) enables rapid development of container-ready microservices, involving very little or zero coding. REST-based microservices can be tested using Swagger or Postman. SOAP-based microservices can be tested using SOAPUI. Container-ready microservices can be built into a Docker image [12], together with the required lightweight operating system, runtime libraries and database drivers. Docker images are deployed and run as lightweight virtual machine (VM) containers within the target cloud environment [12]. Microservices are small enough that DevOps tools such as Jenkins can be used to automate the build, test, and deploy steps [17]. Kubernetes is a popular Docker cluster management suite which covers; service discovery, monitoring, orchestration, load balancing, and cluster scheduling [19]. The microservice development lifecycle, annotated with some popular tools, is shown in Figure 10 below.

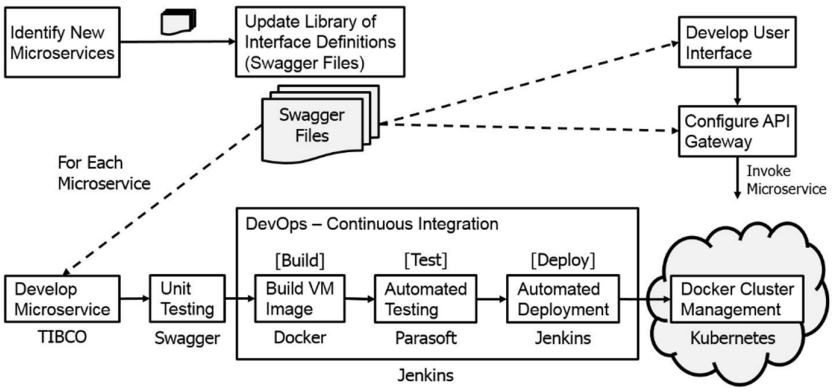

Figure 10 Microservice Development Lifecycle

1.5 Transitioning from Monoliths to Cloud-based Microserves

Migrating from legacy monoliths to a services-oriented architecture has been a long standing challenge [6], up to and including the recent microservices era [3]. In the pre-microservices era, the best outcome of an SOA migration was to provide a layer of abstraction (i.e. a services layer) in front of the legacy monolith, in order

to provide a more flexible architecture while extending the lifespan of the legacy system [21]. There are several case studies of SOA migrations in banking [6, 9].

In the microservices era, the ultimate goal for established enterprises is to replace their on-premises legacy monoliths with a functionally equivalent collection of cloud-based microservices which can be independently developed, deployed, and scaled. Only one case study could be found of monolith to microservices migration in banking [4], and in this case the bank did not de-commission its legacy monolith.

One of the main migration challenges involves reverse engineering of the legacy monolith in order to identify service candidates [25]. If the source code and/or database schema are not available for analysis, capturing and analysing the runtime interaction at the monolith interface (API) can help to identify service candidates, as illustrated in Figure 11 below. Service identification can also be aided by referring to industry models as discussed in the previous section.

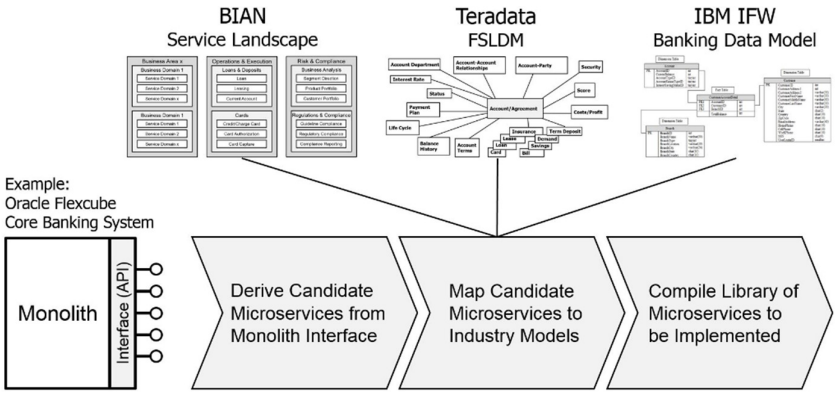

Figure 11 Microservice Identification

Migration Phases

In this section, we offer a phased approach for migrating from a monolith to a cloud-based microservices architecture, as shown in Figure 12 below and detailed in the section which follows. The migration phases presented here are based on an actual core banking system migration conducted in an academic setting under a project referred to as SMU tBank [14], whereby an Oracle Flexcube retail banking system was directly replaced by over 200 microservices.

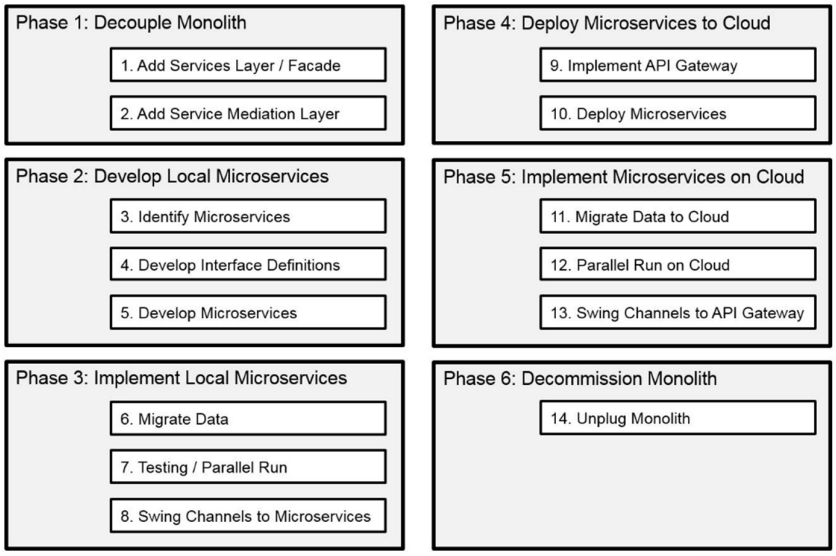

Figure 12 Migration Phases

Phase 1: Decouple Monolith

A common approach for decoupling the frontend presentation layer from the bank-end business logic layer, is to introduce a façade layer [10, 20] between the user interface and the monolith, in order to prepare for the eventual transition away from the monolith. Initially, each façade implements "pass-through" logic (i.e. no data transformation) which reflects the underlying monolith interface, such that any existing user interfaces do not require code changes, and are then physically decoupled from the monolith. To cater for any new user interfaces (e.g. banking channels), each façade may then be refactored into a service which implements the target microservice interface definition, if already identified, such that the service "adapter" performs a data transformation back to the underlying monolith interface. The façade/services layer is illustrated in Figure 13 below.

A service mediation layer [14] is then introduced above the façade/services layer, to provide runtime control over the channel-to-service mapping. For example, if service X invokes the monolith interface for "getAccountBalance", and service Y invokes the equivalent microservice for "getAccountBalance", and both services use the same request/reply fields as specified in the service interface definition, then through runtime control, channel Z can be reassigned to consumer Service Y (microservice) instead of Service X (monolith). With this capability, it is possible to reassign i.e. "swing" the entire set of channels to consume microservices, in one shot, without having to change a single line of code in any of the channels. The service mediation layer is illustrated in Figure 13 below. The services mediation layer also provides monitoring, logging, and security features.

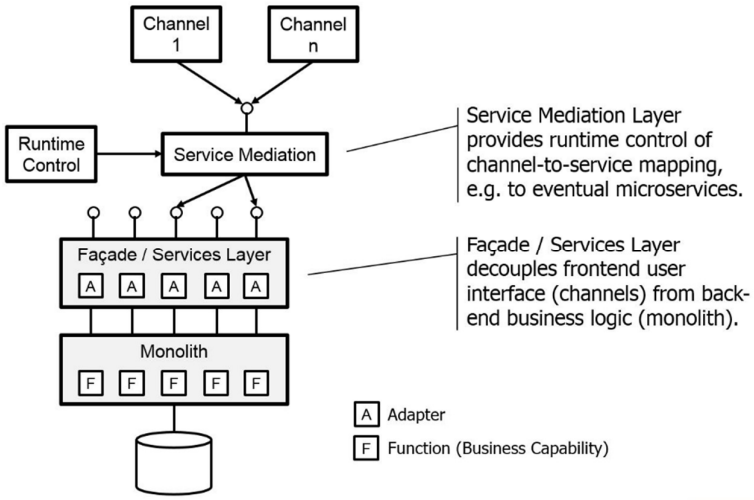

Figure 13 Decoupling User Interface from Monolith

Phase 2: Develop Local Microservices

Decompose the monolith into separate microservices. This may involve reverse engineering the monolith in order to identify candidate microservices [25], as illustrated in Figure 12 above. Service identification is both the most tedious step as well as the most critical step in the entire migration process. It is important to realize the optimum level of cohesion and loose coupling for each microservice.

Develop a library of microservice interface definitions in the form of Swagger files (or WSDL files), which can then be imported into any number of standards compliant microservices development and testing tools. Employ a design time governance tool to manage the microservices design lifecycle, and to make the interface definitions available to developers.

Develop and unit test the microservices, as illustrated in Figure 12 above. The microservice should implement the equivalent business logic and the equivalent data schema as the original function within the monolith. While each microservice can be developed by a small team, a complex monolith such as a core banking system may be decomposed into several hundred microservices. Therefore this is the most resource intensive step in the entire migration process. GUI-driven development, standards-based testing tools, and DevOps continuous integration (build, test, deploy) tools enable rapid development of microservices as illustrated in Figure 10 above.

Phase 3: Implement Local Microservices

Once the microservices are developed, unit tested and deployed locally i.e. on-premises, then the channel-to-service mapping can be changed independently or in batches. For each microservice the following steps are repeated; 1) migrate the data from the monolith to the microservice, 2) conduct a parallel run, such that the

channel invokes both the monolith and microservice, and the resulting data is reconciled between the two, and 3) change the channel-to-service mapping to reassign i.e. "swing" the channel to invoke the microservice instead of the monolith. This process can be repeated systematically, until all of the channels are invoking only microservices. At any point in time, any channel-to-service mapping can be temporarily reassigned back to the monolith, in case of a bug. Service mediation capability enables channels to swing back and forth between the monolith and the microservice without changing any code or configurations on the channel. This capability is illustrated in Figure 14 below (annotations 1 and 2).

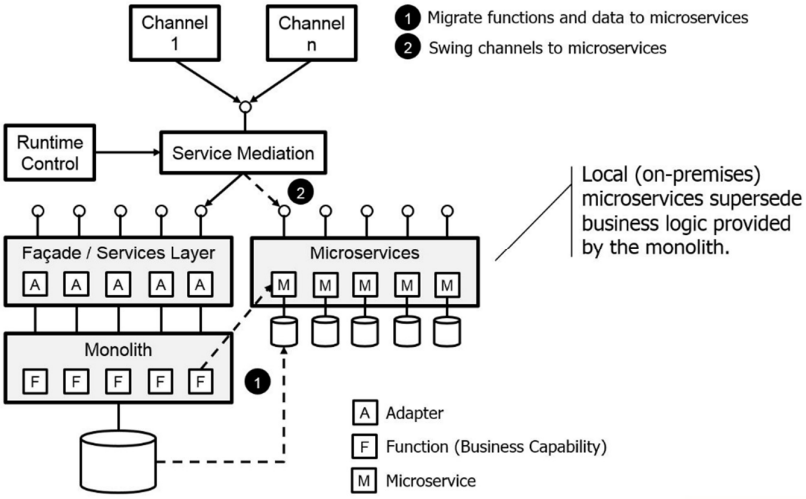

Figure 14 Migrating to Local (On-Premises) Microservices

Phase 4: Deploy Microservices to Cloud

Implement an API gateway in the target cloud environment, to provide a single point of entry and a simple point of control for microservices invocation. Deploy the microservices from the on-premises environment to the cloud environment. Deploy any necessary microservices management and monitoring tools onto the cloud. Conduct end-to-end testing to ensure each microservice can be invoked externally via the API gateway.

Phase 5: Implement Microservices on Cloud

Once the microservices have been implemented locally i.e. on-premises, then the channel-to-service mapping can be changed independently or in batches. For each microservice the following steps are repeated; 1) migrate the data from the on-premises microservice to the cloud-based microservice, 2) conduct a parallel run, such that the channel invokes both the on-premises microservice and cloud-based microservice, and the resulting data is reconciled between the two, and 3) change the channel-to-service mapping to reassign i.e. "swing" the channel to invoke the

cloud-based microservice instead of the on-premises microservice. This process can be repeated systematically, until all of the channels are invoking only cloud-based microservices. At any point in time, any channel-to-service mapping can be temporarily reassigned back to the on-premises microservice, in case of a bug. Service mediation capability enables channels to swing back and forth between the on-premises microservice and the cloud-based microservice without changing any code or configurations on the channel. This capability is illustrated in Figure 15 below (annotations 3 and 4).

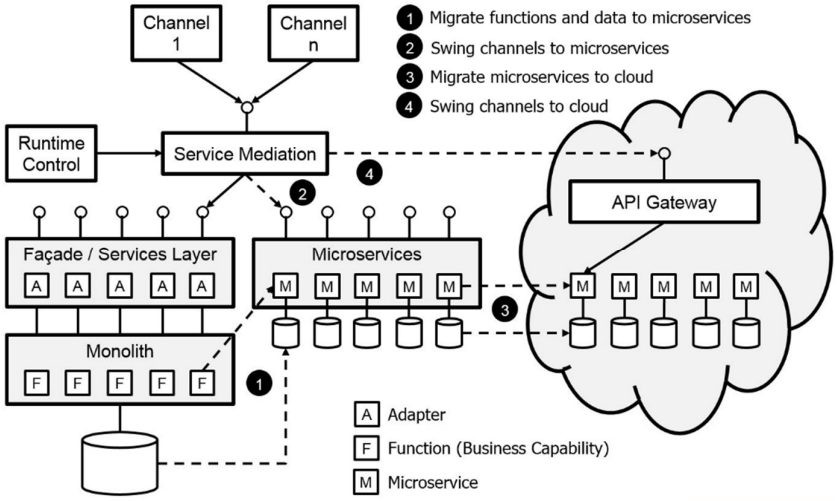

Figure 15 End-to-End Migration from Monolith to Cloud-based Microservices

Phase 6: Decommission Monolith

At this point, or even after Phase 3, the monolith is no longer used and can be decommissioned i.e. taken off line. The on-premises environment then becomes a staging area for microservices development and testing. Channel applications in a UAT environment can be mapped to invoke the on-premises microservices. Existing channels can be systematically refactored to invoke the API Gateway directly, instead of via the Service Mediation layer. New channels and third party apps can invoke the API Gateway directly. The Service Mediation layer would remain until all of the other remaining monoliths, and any future acquired monoliths, are eventually migrated to cloud-based microservices. Figure 16 below shows the final configuration.

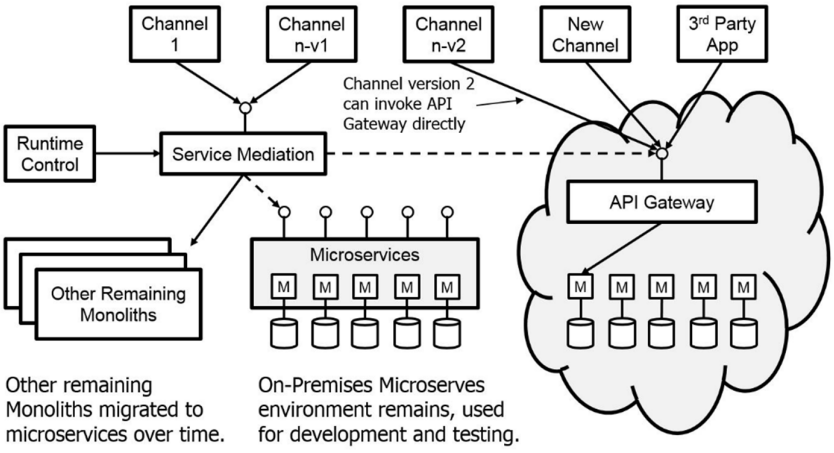

Figure 16 Final Configuration of On-Premises and Cloud-Based Environments

Post-Migration Benefits

For the case of SMU tBank [14] for which the above migration phases are based upon, a number of benefits have been realized as follows:

Performance – Average response time as measured at the service mediation logging point improved from 200ms (monolith) to 40ms (microservice), the difference being the database technology used. With Oracle Flexcube core banking system, we were locked in to using the heavy footprint Oracle database. And for our database intensive microservices, we selected MySQL ndbcluster replication engine which operates in-memory efficiently.

Reuse/Agility – During one stage of the SMU tBank development, three student teams developed four banking channels (Teller, Internet Banking, Mobile Banking, and ATM-simulation) concurrently during one school semester, without creating any new business logic or database tables. This was possible due to their reuse of existing microservices, which were developed during the previous semester.

Collaboration – SMU tBank cloud-based microservices are available for use by other learning institutions. One such institution has used the SMU tBank Open API as the basis for student projects, whereby student teams develop their own banking applications or FinTech alternatives. The SMU tBank Open API has attracted attention from our industry partners. Future work includes collaborating with a large Swiss investment bank to develop a library of BIAN/IFX compliant microservices for wealth management.

1.6 Conclusion

When organizations continue their digital transformation efforts, they should consider an agile style of application architecture which enables the rapid delivery of new cloud-based digital services. Microservices architecture is seen as a key enabler towards this effort. The main tenet of this architecture is to develop software applications more quickly by building them as collections of independent, small, modular services. A primary benefit of this architecture is to empower decentralized governance that allows small, independent teams to innovate faster, thus improving time-to-market of new digital services.

This chapter contributes to the software engineering community by filling a gap in the literature around best practices and methodologies for decomposing monoliths and transitioning to cloud-based microservices. This chapter presented two approaches; a) blank slate approach, whereby applications are developed completely from cloud-based microservices from day one, and b) migration approach, whereby existing monoliths are decomposed into cloud-based microservices, and transitioned function by function onto the cloud, until the original monolith can be literally unplugged. Though the context and examples presented in this chapter relate to the banking domain, the method is generic enough to be applied to other domains such as e-commerce, supply chain and logistics, health care, etc. The blank-slate approach is best suited for building new applications, and the migration approach is best suited for transitioning from existing monoliths to a microservices architecture. The migration methodology presented in this chapter is more detailed compared to the blank state approach. Our future work will focus on further identifying and refining the steps for developing microservices-based enterprise solutions from a blank slate.

References

1. Caetano, A., Silva, A.R., and Tribolet, J., "Business process decomposition-an approach based on the principle of separation of concerns," *Enterprise Modelling and Information Systems Architectures (EMISAJ),* vol. 5, no. 1, pp. 44-57, 2010.
2. Cerny, T., Donahoo, M.J., and Trnka, M., "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review,* vol. 17, no. 4, pp. 29-45, 2018.
3. Di Francesco, P., Lago, P., and Malavolta, I., "Migrating towards microservice architectures: an industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29-2909: IEEE.
4. Dragoni, N., Dustdar, S., Larsen, S.T., and Mazzara, M., "Microservices: Migration of a mission critical system," *arXiv preprint arXiv:1704.04173,* 2017.
5. Frey, F.J., Hentrich, C., and Zdun, U., "Capability-based service identification in service-oriented legacy modernization," in *Proceedings of the 18th European Conference on Pattern Languages of Program*, 2015, p. 10: ACM.

6. Galinium, M. and Shahbaz, N., "Factors affecting success in migration of legacy systems to service-oriented architecture (SOA)," *School of Economic and Management, Lund University,* 2009.
7. Gysel, M., Kölbener, L., Giersche, W., and Zimmermann, O., "Service cutter: a systematic approach to service decomposition," in *European Conference on Service-Oriented and Cloud Computing*, 2016, pp. 185-200: Springer.
8. Indrasiri, K. and Siriwardena, P., "The Case for Microservices," in *Microservices for the Enterprise*: Springer, 2018, pp. 1-18.
9. Khadka, R., Saeidi, A., Jansen, S., Hage, J., and Haas, G.P., "Migrating a large scale legacy application to SOA: Challenges and lessons learned," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, 2013, pp. 425-432: IEEE.
10. Knoche, H. and Hasselbring, W., "Using microservices for legacy software modernization," *IEEE Software,* vol. 35, no. 3, pp. 44-49, 2018.
11. Kohlmann, F. and Alt, R., "Aligning service maps-a methodological approach from the financial industry," in *2009 42nd Hawaii International Conference on System Sciences*, 2009, pp. 1-10: IEEE.
12. Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., and Pallickara, S., "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 159-169: IEEE.
13. Malavalli, D. and Sathappan, S., "Scalable microservice based architecture for enabling DMTF profiles," in *2015 11th International Conference on Network and Service Management (CNSM)*, 2015, pp. 428-432: IEEE.
14. Megargel, A., "Digital banking: Overcoming barriers to entry (Doctoral Dissertation)," Retrieved from Singapore Management University: https://ink.library.smu.edu.sg, 2018.
15. Megargel, A., Shankararaman, V., and FAN, T.P., "SOA maturity influence on digital banking transformation," *IDRBT Journal of Banking Technology,* vol. 2, no. 2, p. 1, 2018.
16. Natis, Y.V., "Core Architecture Principles for Digital Business and the IoT — Part 1: Modernize. Gartner Publication G00324415.," 2017.
17. Palihawadana, S., Wijeweera, C., Sanjitha, M., Liyanage, V., Perera, I., and Meedeniya, D., "Tool support for traceability management of software artefacts with DevOps practices," in *2017 Moratuwa Engineering Research Conference (MERCon)*, 2017, pp. 129-134: IEEE.
18. Pardon, G. and Pautasso, C., "Consistent Disaster Recovery for Microservices: the CAB Theorem," *IEEE Cloud Computing,* 2017.
19. Peinl, R., Holzschuher, F., and Pfitzer, F., "Docker cluster management for the cloud-survey results and own solution," *Journal of Grid Computing,* vol. 14, no. 2, pp. 265-282, 2016.
20. Richardson, C. and Smith, F., "Microservices: From design to deployment," *Nginx Inc,* pp. 24-31, 2016.
21. Shankararaman, V. and Megargel, A., "Enterprise Integration: Architectural Approaches," *Service-Driven Approaches to Architecture and Enterprise Integration,* vol. 67, 2013.
22. Sun, Y., Nanda, S., and Jaeger, T., "Security-as-a-service for microservices-based cloud applications," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 50-57: IEEE.
23. Wikipedia. (2019). *Monolithic application*. Available: https://en.wikipedia.org/wiki/Monolithic_application

24. Wikipedia. (2019). *Microservices*. Available: https://en.wikipedia.org/wiki/Microservices

25. Winter, A. and Ziemann, J., "Model-based migration to service-oriented architectures," in *International Workshop on SOA Maintenance and Evolution*, 2007, pp. 107-110: CSMR.

26. Ząbkowski, T., Karwowski, W., Karpio, K., and Orłowski, A., "TRENDS IN MODERN BANKING SYSTEMS DEVELOPMENT," *INFORMATION SYSTEMS IN MANAGEMENT XVI,* p. 82, 2012.

Index