S	
E1	
E2	
For Office Use Only	

Masters Project Final Report

(MCS)

2019

Project Title	Process of converting monolithic application to microservices-based architecture
Student Name	Thiwankan C. Kapugama Arachchi
Registration No. & Index No.	16440068 / 2016MCS006
Supervisor's Name	Prof. K. P. Hewagamage

For Office Use ONLY

Process of Conversion Monolithic Application to Microservices Based Architecture

A dissertation submitted for the Degree of Master of Computer Science

T. C. K. Arachchi
University of Colombo School of Computing

2019

Declaration

The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: Thiwankan C. Kapugama Arachchi

Registration Number: 2016MCS006

Index Number: 16440068

____

Signature: Date:

This is to certify that this thesis is based on the work of

Mr./Ms. T. C. Kapugama Arachchi

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Prof. K. P. Hewagamage

____

Signature: Date:

Table of Contents

Table of Figures

Abstract

Software architecture is the basic structure for software application development. Monolithic software architecture is used to develop software products in earlier decades. Microservices is recently introduced software architecture that promises of high maintainability and scalability. Most of the legacy applications are followed by the monolithic architecture pattern. Software complexity becomes unmanageable with the growth of the code base of the application. Monolithic architecture fails when the complexity of software increases.

To overcome difficulties of the monolithic architecture, computer scientists try to move for better software architecture such as microservices architecture. The transformation of a real-world operational software application into a new architecture-based application is a challenge. Transformation of monolithic based application into the microservices-based application has been discovered in a theoretical manner. Identify theories and concepts related to the transformation of software architecture will be one concern of the thesis. Then based on analysis will evaluate the transformation approaches.

Based on the evaluated studies, three processes will be introduced to convert, the monolithic real-world operation application into microservices. “Refactoring from scratch”, “incremental refactoring” and “convert only new components into microservices” are the three processors that will be described. Introduced conversion processors are in a conceptual manner which can directly apply to real-world application. Since the conversion processes are in a conceptual manner, it can optimize based on the situation and requirements.

The research outcome evaluated by applying the conversion processes to the real-world application (Sri Lanka Telecom Work Force Management System). Evaluation of the processes based on the number of microservices generate, time to complete conversion and resources used for conversion. Outcome of the research is about three conversion process which convert monolithic based application to microservice application.

1. Introduction

1.1. Background

Software and Information Technology has distributed over different domains. Many software solutions can be found on market to fulfill customer expectations. Information technology is a vast domain that changes continuously. Computer scientists release a variety of expertise development tools, frameworks, programming languages, concepts with improvements. Increasing customer expectation and demand is the reason for the continuous evolution of information technology. The evolution of software development improves tools and techniques by introducing new features and resolve defects in previous versions. New software frameworks, design patterns, software architectures, and other techniques improve the quality of the software and reduce the effort to maintain software products. Even for software development tasks such as design software architecture for an application, there can be different ways to attend and complete the task by using various tools, frameworks, and techniques. However, the challenge is to select appropriate tools, based on task nature. The appropriate tool for a selected task will optimize the development process which helps to achieve customer requirements and expectations. In some cases, the selected tools may seem to be appropriate for the selected situations without any contradiction, however, the selected tools may fail to achieve long term expectations of stakeholders. Therefore, selecting appropriate tools for development is a challenge. Wrong decision-making reason to software crisis, which affects vendor's organization operations such as increase the workload for rework and fixing issues within a tight schedule which reason to produce poor quality software product. Poor quality software will be unsatisfied customers and users. Therefore, the software designing phase is important because it's the place where the user selects the tools and techniques for development. Short term consideration of the situation most of the time the reason for failure.

[1]Thinking pattern related to software development is another important phenomenon. Changes for software engineering reason for different thinking pattern from different parties. For easier the development process it has been introduced dev tool, can be taken as example for changes to software engineering based on thinking pattern. Evolution of thinking pattern reason to improve software application but also invent innovative ideas. For example, previous software engineering solutions always follow the same development process for different client requirements. Traditional server application development used to implement all functionality under one module which may interact with the relational database (monolithic thinking pattern). In the modern, development approach focus has been changed to push business logic from the server-side to the client-side. [1]As an example; nowadays different JavaScript frameworks publish with more capabilities compared to pure JavaScript, such a framework is Angular. Software persistence methodology has been improved to satisfy the important requirements such as unstructured data for enabling dynamic behavior of data which easier to manage. **[2]**NoSQL database is an unstructured database that differs from the relational database. **[3]**The new era always comes up with a technology solution for differ challengers. Scalability, security, reliability, and availability are some of the challenges among them.

Software architectures used to implement a blueprint of software application. Software architecture is about a conceptual plan which differs from one architecture to another. Microservices architecture has been promoted using high maintainability and scalability. As mentioned in different sources, it brings a bunch of advantages and benefits compare to disadvantages or disabilities it owns. Microservices are more suitable for large enterprise applications that need high scalability. The current software industry focuses on higher maintainability, other than scalability. This thesis outcome will be applied to real-world situations to evaluate and confirm that objectives are reached or not.

1.2. Problem

The software development process is an important process in software firms and organizations. Even in small-scale firms try to follow software development process since good software development process, supports to release quality product for the end-users. With technology evolution, the software development process is needed to improve periodically. [4] The outdated software development process effects software industries in several ways. For example, an outdated software development process structure makes it difficult to adopt the evolution of technology. Since the productivity of the organization always follows an extra amount of work. Learn from previous experience and knowledge, it enhanced the way software developed, for example, to deploy a new version of software need more resources to test and verify deployment. DevOps is a solution to overcome resource costs. Therefore, upgrading the software development process is not an overhead task. It will benefit to improve the productivity of the organization. If an organization follows outdated technology within the organization, the employee's knowledge has been limited to a specific domain area in software development. That will cause to hinder the ability of employees.

In the software development life cycle, maintenance is the phase that takes a considerable amount of time compared to other phases in the software development life cycle. Therefore, as a best practice, when developing software, it should try to reduce the maintenance work. At the beginning of software development, it seems the best choice is monolithic architecture since it has a clear start-up and manageable. Growth of software application codebase, reason for increase complexity. Extending functionality, adding new modules or components major reason for codebase growth. If changes have been applied without having proper management, it will cause to increase the complexity of the application. As a result, the customer becomes unsatisfied, since the product not delivered on time. With the complexity of software, it may need rework and testing, which takes more time to provide a solution requested by the customer. It is important to keep track of the modifications in the software application. When it applies different approaches to solve the complexity of a software application, it may become reduce the quality of the software. “Spaghetti code” issue is more common in monolithic architecture-based software which is unstructured and difficult to maintain. The worst-case scenario is if a task takes more time above the realistic time to perform modification can affect the software development timeline. Anyhow, the developers need to concern all replicated code scenarios which consume more time.

Most of the time, the legacy system consists of constrains for a specific technology stack. For example, some of the bank application core still in the COBOL programming language. Even new and innovative technology has been released, it may still hard to integrate with new applications. Vendor and user suffer from technology lock-in (Software developed based on specific programming language and tools). It is difficult for software vendors to move to a new solution to overcome technology lock-in. The legacy system's business domain can be more complexed with different workflows. If users familiar with the legacy application, there are organizational and technical challenges. From the customer's perspective can concern if the current legacy system replacement with a new product, needs more budget and doubt about the new system will meet the performance as the legacy system function. From the vendor's perspective, still, it costs to maintain. Since the legacy system limited to a specific technology, the vendor has to allocate a specific team for product maintenance.

As mentioned earlier, the evolution of software engineering is introduced techniques, tools, and processes to easier software development. The improvement or advancement provides new features for the application. For example, while developing the software can take decisions regards the way to distribute application by using service discovery concepts (DevOps). When it compares upgraded features and previous versions of tools, a large number of different improvements introduced to automate software development. Recently introduced software architecture or design concept that addresses more concerns may important for the long run rather than repeated failures. It will be easier the development and avoid re-invent solution which is already solved. [5] Most of the organizations follow the process re-invent with or without awareness that will cause to higher the cost of product/change and fail the management of software development. However, the major problem identified is that software vendors keep update with software evolution and get advantages from them.

1.3. Objectives

The main objective of the research is to introduce software architecture transformation processes, which convert the monolithic application into a microservices-based application. As mentioned in the problem section, monolithic architecture consists of many drawbacks compared to microservices architecture. On the other hand, microservices inherently support for latest technology and it provides high maintainability. Within the main object, it covers secondary objectives which are, support for further enhancement of architecture conversion concepts such are, reduces rework and test for each modification, enables to use of the latest technologies and tools and motivates to create changes and accept organizational and technical challenges.

From a high-level view of point, the research covered up the most common general issues which happen in the software industry. Most of the legacy systems have been considered as monolithic based architecture. Another important objective is to motivate the software organizations to transform from legacy systems to new microservices architecture. Provided solution process, can be apply to organizational legacy software develop process.

To overcome drawbacks in monolithic architecture, it needs to go for a new solution. This research provides a solution as a microservice-based software application that a new way of development. The architecture transferring decision depends on the technical and organizational decisions. Since the conversion of application consumes more time and budget, it has become challenge. Not all legacy systems need to break down into services, but it needs to identify when to go for microservices. Some legacy systems may operate as expected, therefore no need to go for microservices. Conversion process application knowledge will support to enhance to think beyond the solution provided by this research.

Once all the outcomes are provided, then consider the practical application of the above concept. When applying these outcomes, it will be tried to analyze the challenge and adjust according to the situation. Application knowledge will be provided general way from analysis of application which may more convenience and applicable for any management related software.

In the evaluation, it will be implemented the solution derived from monolithic to microservice conversion. The purpose of the implementation is to point out the start point to the reader. Microservices can adopt and implement different ways that follow microservices concepts. This section is better for software developers who interested in grasp technology related to microservices. On this objective, it’s trying to discuss more detail about best practices which will guide for newcomers.

1.4. Scope

Software architecture is a vast domain in the IT industry. When it comes to microservice architecture and monolithic architecture patterns, the structure of the application completely differs from one to another. Although domain of the thesis still contains a large domain, because architecture concepts can study different ways based on thinking patterns. For example, monolithic architecture is about a single code base as many knowledge sources provide. Someone can develop application which breaks down to layers. In some situations, an application can use a combination of architectural pattern. For example, the monolithic architecture can combine with other architecture patterns such as layered architecture. This thesis mainly concerns about the microservice pattern. However, there will be a brief introduction about both monolithic and microservice architecture at appendices and it just provides high level and very important information for quick reference only.

For non-technical readers can refer to the appendices section for more details relate. It will be easier to get a startup. Beginner who doesn’t have knowledge needs to clear view of the difference between monolithic application and microservice. At some point, it will become overhead to system breaks down into microservice. It needs to clear purpose or reason for architecture transformation. It will help to analyze the solution without having confusion and non-technical reader can understand the idea before going to the technical discussion.

Different researches about the conversion of monolithic to microservices can be found. Anyhow the phenomenon is that it provides all about concepts/patterns proved in a theoretical way. Selected research papers have been presented in briefly. Since implementation depends on the situation, thinking pattern and decision of the organization, this research will provide conversion processes more general way that can address any situation. While the conversion process applies to a real-world situation, the difficulties can be found on the application. Suggestions, solutions, and recommendations to those difficulties will be addressed in this thesis. Application of the introduced processes can be for a general situation like e-commerce related software. Anyway, it can further research for different software such as real-time application.

The application of the three conversion processes, mentioned in this thesis is based on a real situation. Application (Sri Lanka Telecom WFM system) related complete knowledge capturing and documents will not be mentioned in detail on this thesis. For readability/space and time constraints, it is going to discuss two microservices design and implementation only. Within two microservices, it will be explained difficulties faced, and solutions have been taken. The technology and tools for microservices are described in brief. However, the information will motivate to experiment on processes and further enhance findings.

The outcome of the research will be identified as the best method for refactoring and provide guidance to change the monolithic application into microservices. Some situations will be failed to get a better solution since depends on the thinking pattern. Outcome won't be limited to a converting tool or framework and it completely depends on the situation.

1.5.Thesis Application

Sri Lanka Telecom Work Force Management System (WFMS) is an internal application. The main purpose of WFMS system is to communicate with SLT core system and retrieve telephone faulty which added by SLT customers. "Telephone faults" are managed by WFMS regional officers in assigning faulty to allocated workgroup consists of field officers. Once faulty is cleared and completed, final information goes to sync with SLT core system. Several flows are handled within WFMS and this system is spread everywhere in the country as mobile applications and web interfaces that connect over 1000 workers in Sri Lanka Telecom. Currently, the application is based on a single monolithic application that provides the service since 2005. Field officers interested in WFMS product usage. Therefore, WFMS has become a major internal system of SLT. From the vendor aspect, this project needs frequent changes and modifications to fulfill client requirements and fix issues. This system manages SLT connection faulty reported by customers. This information needs to sync with the other internal systems in real-time and it maintains more than 10 inventories for the business process. This thesis is going to apply theoretical concepts to real-world microservice applications.

1.6. Thesis Outline

This thesis consists of five Chapters. The first part is an introduction to the thesis which is mainly highlighted the problem domain and a place where to apply the outcome of the thesis and objective of the research. The second chapter of the thesis is explained in the literature review and it describes the reason for a uses microservice architecture, conversion theory based on the categorization of the situation and generate microservices based on entry points. The third chapter is described as solution processes and a way to apply it to a situation. It is clearly described the problem domain and list of the problem which motivates for the project. In the fourth chapter, it is described as proposing solution details. Under this chapter, in further it discussed different ways to convert monolithic to microservice architecture. Along with a method for documentation of services and best practices involved within it.

2. Literature Review

The following chapter is highlighted in the literature review that used to derive concepts. The problem is explained at the beginning and researches need to find a solution path with findings. Initially, it is described the reason to select microservices instead of other architectures then defined about the researcher's findings that are used to convert into microservices.

2.1. Reason for Use Microservices Architecture

Since this thesis only focuses on the microservices architecture other than any other software architecture patterns. When considering the interest in microservices, it shows dramatically growth.

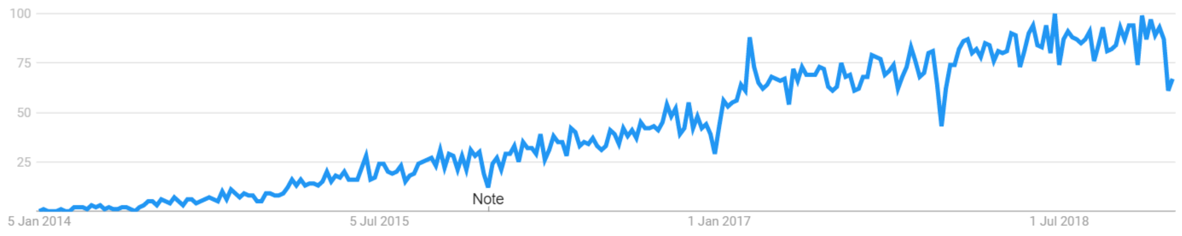


Figure 1 - Microservice interest over time (source: Google Trends)

[6]There will be many facts available when searching about the popularity of the microservices. Recent technology aligns with microservices concepts. Microservices is a concept to implements small services. Derived microservices should independent from each other. Therefore, it has been allowed to use any technology for implements microservices which is not needed to consider old or new technologies. Microservices architecture creates a place to combine the latest technology that arises in this era. As an example; the cloud computing concept is found recently but the usage of cloud computing has become popular along with the microservices system. In recent decades, the customer is expected the high availability and scalability of the application and these expectations can be easily fulfilled by using the microservices. With new business concerns, it may be needed to slight update on microservices or in the worst case, it may add new service to cover up new business concerns.

Introduced conversion processes are solution to the limitation of Service-oriented architecture (SOA) architecture: SOA is a method of designing software and most related to microservice architecture. [8]From another perspective that microservices are another version of SOA. Enterprise project invests in SOA related architecture; however, it contains monolithic behavior. According to business concerns, it is separated into several services. Enterprise Service Bus is used to define business flow and service registration. It's create a bottleneck between ESB and service invokers. It can be communicated with API gateways by using the microservices architecture which is independent of another API.

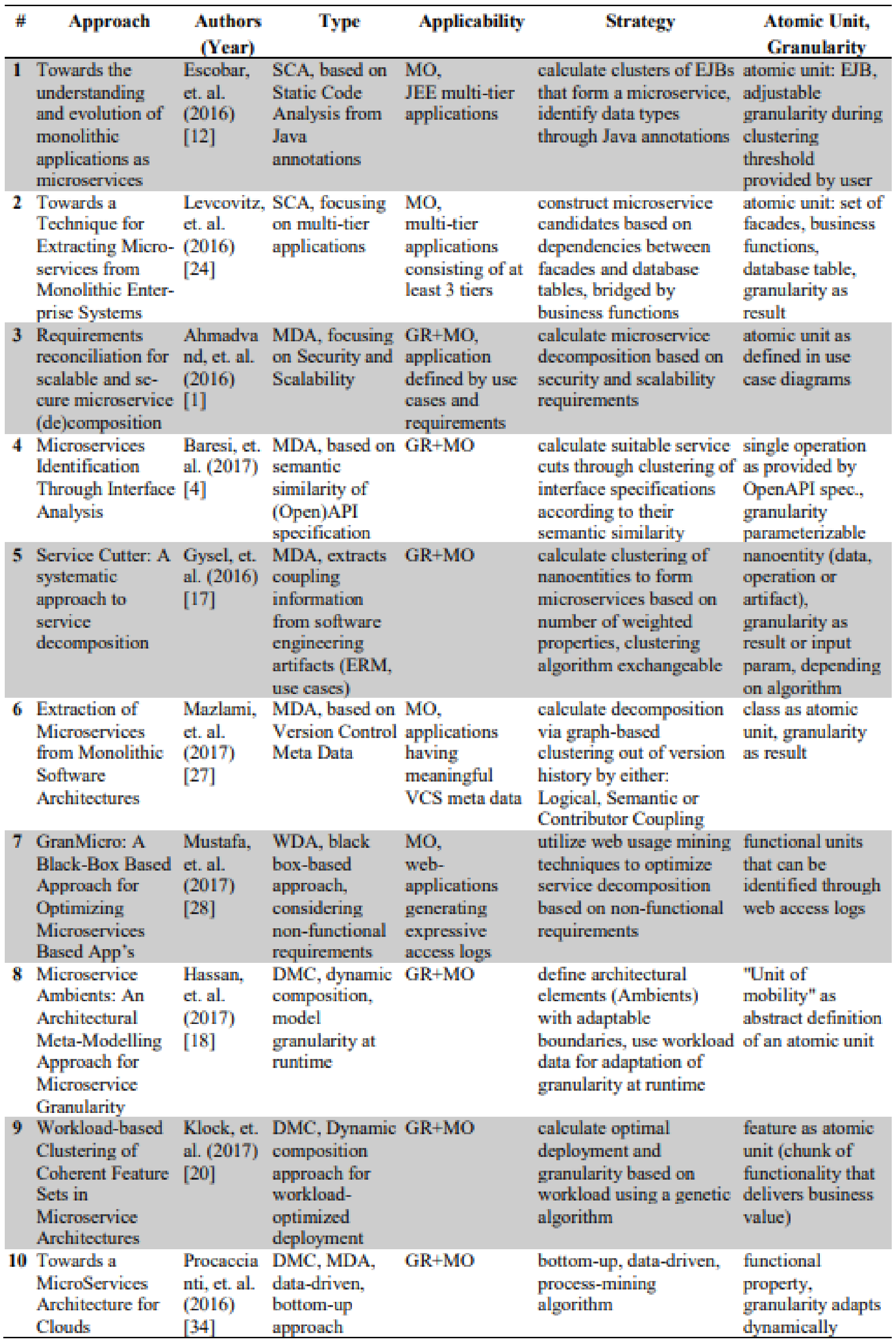
Adoption of best practices: In recent decades, it is introduced several important concerns related to software development. It should manage development concerns and deployment of product against versions to easier the maintenance period. DevOps concepts are derived with combining development and operation task which supports to easier the development and deployment. DevOps is made continues integration process and continues deployment is more important to enhance productivity. As a positive side of microservices, it can be used as best practices for the test environment and production environment.

As mentioned earlier, this thesis is mainly focused on two software architectures, monolithic architecture and microservices architecture. Microservices architecture is a more suitable solution to reduce the difficulties in a monolithic architecture. Some solutions which include the microservice architecture is used to reduce the complexity of the application and improve maintainability (complex application divide into small manageable services) of the application.

2.2. Monolithic to Microservices Conversion Classification Approach

[4] Classification approach is mentioned that, even though many resources are available for finding the guidance to refactor the legacy application, it is difficult to find an appropriate way to refactor. Because refactoring process is costly with overall process implementation and manage team structure. Therefore, it is needed to identify and selects most suitable refactoring strategy. In this thesis, it provides a way to search researches then analyze 10 research publications which provide information to refactor legacy application to microservices. Most important and useful task is that refactoring approach classified based

on decomposition techniques and then provide visually present guidance to identify which decomposition method is suitable for situation.



#	Approach	Authors (Year)	Type	Applicability	Strategy	Atomic Unit, Granularity
1	Towards the understanding and evolution of monolithic applications as microservices	Escobar, et. al. (2016) [12]	SCA, based on Static Code Analysis from Java annotations	MO, JEE multi-tier applications	calculate clusters of EJBs that form a microservice, identify data types through Java annotations	atomic unit: EJB, adjustable granularity during clustering threshold provided by user
2	Towards a Technique for Extracting Micro-services from Monolithic Enter-prise Systems	Levcovitz, et. al. (2016) [24]	SCA, focusing on multi-tier applications	MO, multi-tier applications consisting of at least 3 tiers	construct microservice candidates based on dependencies between facades and database tables, bridged by business functions	atomic unit: set of facades, business functions, database table, granularity as result
3	Requirements reconciliation for scalable and se-cure microservice (de)composition	Ahmadvand, et. al. (2016) [1]	MDA, focusing on Security and Scalability	GR+MO, application defined by use cases and requirements	calculate microservice decomposition based on security and scalability requirements	atomic unit as defined in use case diagrams
4	Microservices Identification Through Interface Analysis	Baresi, et. al. (2017) [4]	MDA, based on semantic similarity of (Open)API specification	GR+MO	calculate suitable service cuts through clustering of interface specifications according to their semantic similarity	single operation as provided by OpenAPI spec., granularity parameterizable
5	Service Cutter: A systematic approach to service decomposition	Gysel, et. al. (2016) [17]	MDA, extracts coupling information from software engineering artifacts (ERM, use cases)	GR+MO	calculate clustering of nanoentities to form microservices based on number of weighted properties, clustering algorithm exchangeable	nanoentity (data, operation or artifact), granularity as result or input param, depending on algorithm
6	Extraction of Microservices from Monolithic Software Architectures	Mazlami, et. al. (2017) [27]	MDA, based on Version Control Meta Data	MO, applications having meaningful VCS meta data	calculate decomposition via graph-based clustering out of version history by either: Logical, Semantic or Contributor Coupling	class as atomic unit, granularity as result
7	GranMicro: A Black-Box Based Approach for Optimizing Microservices Based App's	Mustafa, et. al. (2017) [28]	WDA, black box-based approach, considering non-functional requirements	MO, web-applications generating expressive access logs	utilize web usage mining techniques to optimize service decomposition based on non-functional requirements	functional units that can be identified through web access logs
8	Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity	Hassan, et. al. (2017) [18]	DMC, dynamic composition, model granularity at runtime	GR+MO	define architectural elements (Ambients) with adaptable boundaries, use workload data for adaptation of granularity at runtime	"Unit of mobility" as abstract definition of an atomic unit
9	Workload-based Clustering of Coherent Feature Sets in Microservice Architectures	Klock, et. al. (2017) [20]	DMC, Dynamic composition approach for workload-optimized deployment	GR+MO	calculate optimal deployment and granularity based on workload using a genetic algorithm	feature as atomic unit (chunk of functionality that delivers business value)
10	Towards a MicroServices Architecture for Clouds	Procaccianti, et. al. (2016) [34]	DMC, MDA, data-driven, bottom-up approach	GR+MO	bottom-up, data-driven, process-mining algorithm	functional property, granularity adapts dynamically

Figure 2 - Overview of 10 Decomposition Approaches (Source: monolithic classification [4])

As mentioned in above table (Figure 2) it has been used ten research papers related to acquire to derive conversion processes. Classifications of migration approaches are static code analysis which scan application source code, meta-data aided approach which use architectural description (4 in 1 diagram, historical data) to decomposition, workload data-based approach which analysis application module wise data usage and do the migration and finally dynamic microservices decomposition which based on behavior of runtime of application uses for decomposition of microservices. For an example no 1 approach is analyzed monolithic application using EJBs and data type using Java annotation then convert each EJB into microservice. No 1 approach classified under static code analysis. Likewise, it provides a situation that needs to match with the current situation in hand, then it provides a solution approach to convert into microservices.

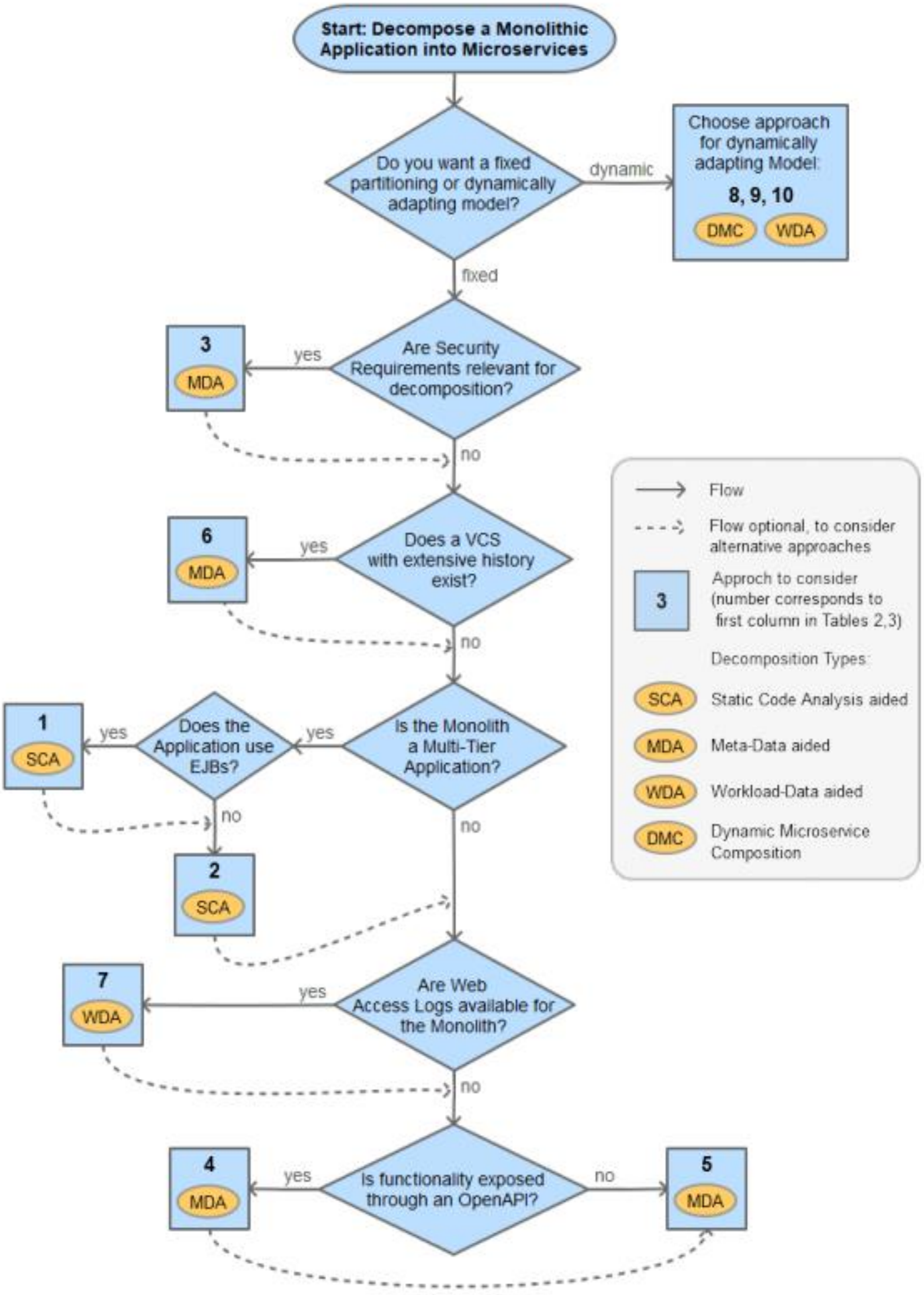

Start: Decompose a Monolithic Application into Microservices
Do you want a fixed partitioning or dynamically adapting model?
dynamic
Choose approach for dynamically adapting Model:
8, 9, 10
DMC
WDA
fixed
Are Security Requirements relevant for decomposition?
yes
3
MDA
no
Does a VCS with extensive history exist?
yes
6
MDA
no
Is the Monolith a Multi-Tier Application?
yes
Does the Application use EJBs?
yes
1
SCA
no
2
SCA
no
Are Web Access Logs available for the Monolith?
yes
7
WDA
no
Is functionality exposed through an OpenAPI?
yes
4
MDA
no
5
MDA
Flow
Flow optional, to consider alternative approaches
3
Approch to consider (number corresponds to first column in Tables 2,3)
Decomposition Types:
SCA
Static Code Analysis aided
MDA
Meta-Data aided
WDA
Workload-Data aided
DMC
Dynamic Microservice Composition

Figure 3 - Decision Guide for Decomposition Approaches (Source: monolithic classification [4])

Finding exact approach for decomposition is always not possible. Failures can happen on selected approach. Therefore, it is recommended to select multiple approaches as an alternative and it depends on the result situation. The above diagram describes an approach to select and validate. If the selected approach is failed, it can be tried out another approach as mentioned in flow diagram. For example, statically analyze the monolithic architecture then try to come up with the model. Model can be dynamic or fixed and based on the model, it tries to evaluate several factors such as non-functional requirements (security concerns), version control status, tire of architecture, bean type likewise.

When consider about above research paper, few limitations can be highlighted. Decomposition guidance of the above approach is needed for the software engineer's knowledge to capture and follow guidance. One drawback is identified, which discussed research papers are difficult for refer by person who non-technical. Another phenomenon is that cannot be ensure about obtaining accurate microservice after decomposition. Categorization of researches as discussed, may not always success due to differentiation of content. As conclusion, most of the time, the output relevant approach is a static code analysis scenario.

2.3. Using Microservices Legacy Software Modernization

Microservices maintainability is motivation for legacy modernization research paper motivation to introduce conversion pattern. Based on previous experience authors tries to express decomposition pattern by guidance through application. The primary driver for modernizing the monolithic application is the fact that it has become increasingly difficult to deliver new features on time. Since a large, high-priority project requires fundamental changes to large parts of the application, this lack of evolvability is considered a strategic risk. According to the developers, there are two main reasons for this low evolvability—namely, a deterioration of the application's internal structure and the high number of entry points for monolithic application. This has made the impact of changes difficult to assess, leading to a high amount of testing and rework. Secondary modernization drivers are the vendor and technology lock-in as well as the fact that many developers are close to retirement and language specific developers are difficult to obtain.

Guidance for the decomposition of legacy application into microservices provided as several steps. The first step of the modernization process is concerned with defining an external service facade that captures the functionality required by the monolithic systems in the form of well-defined service operations. The implementation of these operations is performed in later steps. The major challenge of this step is to define domain-oriented services that provide the functionality needed by the clients, but without conserving questionable design decisions from the legacy application. Authors approached this challenge from two sides. First, They created a target domain model and used it to define service operations from scratch that expected to be provided by the application. Afterward, employed static analysis to identify the “entry points” of the application. Entry point can be program, method or database related table. By using analyzed entry point try to invoke functionality and formulate services. Similar and redundant operation merged for reduce complexity. Important of this process that identify way to replace existing entry point with new services.

Second step about implementation of identified services by use of entry points. Implementation adopt existing system, any how without integrate since it become more risk to do this task at same step. Challenge of this step to identify service integration that fulfill functionality identified in step one. Due to optimization and refinement, some services need to develop from scratch. Each implemented service need to test properly by using different testing techniques (Unit test). In some case it cannot be test legacy environment due limitation that insufficient for proper testing (difficult for create mocking object). Therefor it needs to use test environment suitable for proper test. Cloud based environment is recommended by author (Docker).

Once the service operations are implemented, as third step monolithic application can start migrating to the new facade by replacing their existing accesses with service invocations. This step poses organizational as well as technical challenges and usually consumes a large part of the overall project time and budget, since large parts of the monolithic application must be changed and tested. In order to support the development teams during the migration, author mentioned that a transition documentation. This documentation contained a textual description of how to replace each of the entry points identified in step one with one or more service operations. For each of the new operations, detailed descriptions and code snippets were provided to facilitate the transition as much as possible. This documentation was considered very helpful by the developers.

In step four establish external service façade using service implemented. Inspect appropriate program and functionality need to perform this step which is identified through step one to three. Due to risk and resource demanding this step separate from last discussed steps. Step five is last step convert all services to microservices. Once all desired service façades are established, the process of introducing microservices can begin, as the adapted service implementations can now be transparently replaced. It is, however, important to note that several modernization goals have already been reached. Although the implementation is still based on the old technology which use for monolithic application, it is now only accessed using well-defined, platform-independent interfaces, and the application has been internally restructured into the desired components.

To conclude the literature review section, it has been discussed better conversion architecture suitable for replaces with a monolithic architecture. By the research, it identified that microservice architecture provides solutions for drawbacks found within monolithic architecture. Therefor decided to select microservice architecture for output architecture. Once result in architecture determine, research about conversion that ends in microservices. As describe in second section it mentions that there is classification of research paper used to convert into microservice. However, it better starting point for designing general process that need to applicable for most of the scenario. In second research provide complete system decomposition in with proper plan. Any how it mentions that complete its process take considerable time and need more effort to analyze whole monolithic application. Then it discusses reason for latest organization adopt microservices for development. From the mentioned literature review it help to find and design conversion process that applicable for any general situation in e-commerce application.

3. Methodology

In this section, discuss main outcome of the thesis which that monolithic to microservices conversion, it will be considered different ways to perform monolithic to microservices decomposition. At first, it will highlight the design approach. Thereafter in detail discussion of each approach considered finally implementation information will discuss.

3.1. Plan for migration to microservices

[7]Before going for the migration there must be reasonable pitfalls in the existing system. Note that without considering facts to migration to monolithic is overhead of work. At some stage, the monolithic system will be provided the following symptoms that it cannot perform well in further and you must go for microservice architecture. It will be discussed some of the difficulties in the monolithic application such as complexity. Complexity of application is related to the size of the application. At the beginning of the application can see as monolithic architecture can manage and appropriate selection. However, system changes reason to customer demands and other facts such as technology evolution. By applying changes to system reason to grow. On the other hand, it effects to development best practice which documentation and manage changes to application. Unmanaged changes affect development, re-invent the wheel and less re-usability reason for spaghetti code. Finally, it difficult to apply changes, the number of bugs increases with few changes, takes time to deliver changes and in the worst customer unsatisfaction happens.

When the difficult situations arrived from the monolithic system, apply patches and rework becomes difficult and won’t achieve what to expect. Apart from the above concern, it may be a reason that new non-functional request which affects to the whole module of the system such as enforce security concerns, to enable system high availability and scalability and so on. First need to carefully decision making for the urgent task to convert into microservice.

Documentation of the system is an asset to the software system. Anyhow, clarity of information within the document can make better decisions. Therefore, documentation should not outdated or invalid for analysis. At some point documentation of software easier the conversion task. The following section will list down different approaches of monolithic to microservices migration.

3.2.Conversion Plan 1: Develop from Scratch

By its name, it can understand this approach. Begin from basic and grow up microservices without considering the monolithic system. This approach tries to re-structuring legacy system that builds a system plan from start and does refactoring that a new system needs to build with functionality that provides from the legacy system. When refactoring into a microservices-based system it's important to follow microservices principles that small business concerns address by one service. As a result, it may outcome different entry points that can be open which not available in the legacy systems to improve reusability.

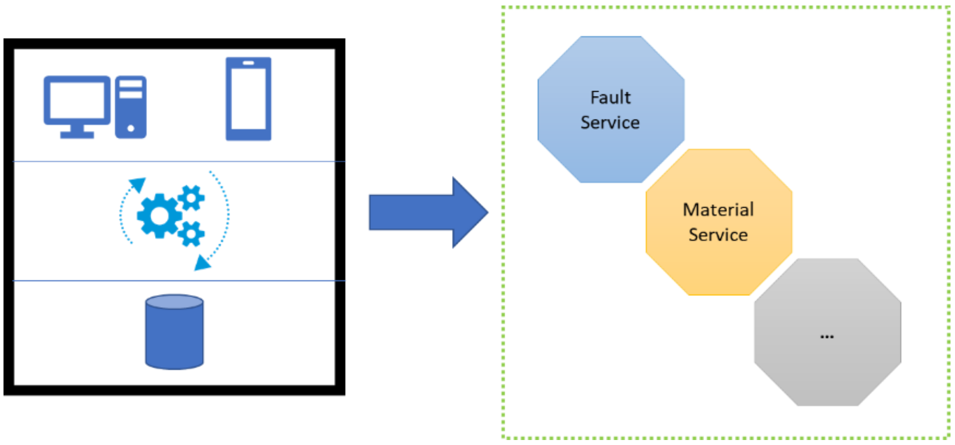


Figure 4 – Develop microservices from scratch to existing legacy system

When considering advantages of this process which migrating to microservices, it can identify existing defects or performance issues and new changes which may be extend existing functionality or add new module can be implemented along with migration.

Anyhow this conversion comes with a cost, which means many drawbacks can be found. It may legacy system large and very complex. In that scenario analyze legacy system trivial task. Lack of system analysis can lead to build-up a system that won't meet customer expectations or build a functionality that not related expectation or it may mislead the existing functionality. With the spec of the legacy system, it not easier to find a place to start the migration, because there can be several dependencies all over the system. Some of the drawbacks consists of this flow. Since the entire module converted into microservice based on monolithic application deviation can happen compared to original business workflow. Some changes can affect positively (which defects can be identified and provide the solution for them) but even though negative effects (introduce new defects into workflow or missing operations in workflow) also need to expect. Until the end of the refactoring cannot see the output of the process. The only evidence that will be tracked project progress can estimate based on the design plan. Resource allocation for the implementation of project vary depends on priority and time allocated for the entire project. Some of the implementations may need many time tests in different scenarios and need to do rework based on requirements.

3.3.Conversion Plan 2: Adopt Microservices for New Functionality

There can be a situation the legacy system performs well as expected. However, based on customer requirement try to extend the system with new feature functionality become difficult. In this kind of situation, it is better to use this approach. Specialty in this approach is that It won't affect to root monolithic system. Therefore, new features/changes will develop and deploy as microservice. The basic idea of separation is that isolate new functionality to easier the development and introduce logic separation. As outcome of this process it can be found two system one based on legacy and other based on microservices architecture. With growth of system microservices based system populate than legacy system.

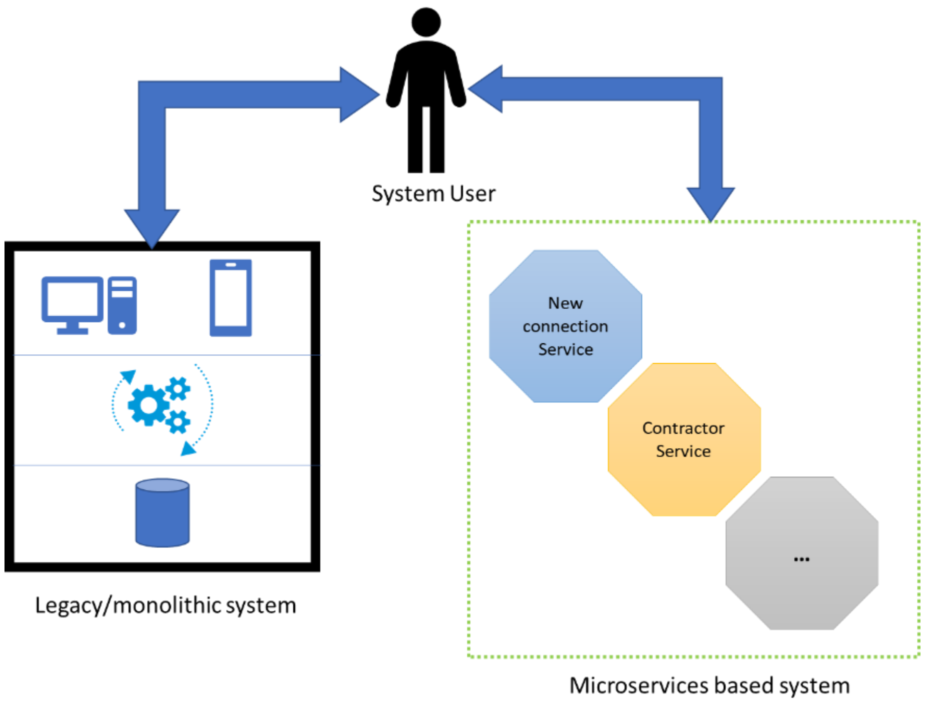


Figure 5 - Add new feature/functional as microservice

In this case, the most recent feature will be integrated with microservices architecture. From that, it would avoid the growth of the monolithic system code base and easier to manage. New microservice apply along with legacy can scale independently and deliver separately to the old module. By adopting this approach, the development team makes the new systems easier to modify future where it is not the case of monolithic replace with another.

3.4. Conversion Plan 3: Replace Functionality with Microservices

This approach very similar to the above-mentioned conversion plan 2. However, specialty in this situation that it needs to replace the monolithic core application with microservices. In this scenario, it is easier to think about the monolithic system as a large service and need to decomposition this large service that becomes a set of microservice. It tries to identify business domain-specific functionality which is can manage with few effort and team. Then identified functionality will be moved to microservice without changing process flow. Finally, in the legacy monolithic system modify with glue code which unplugs functionality implementation in the legacy system and changes entry point to use microservice instead. The section will provide more information.

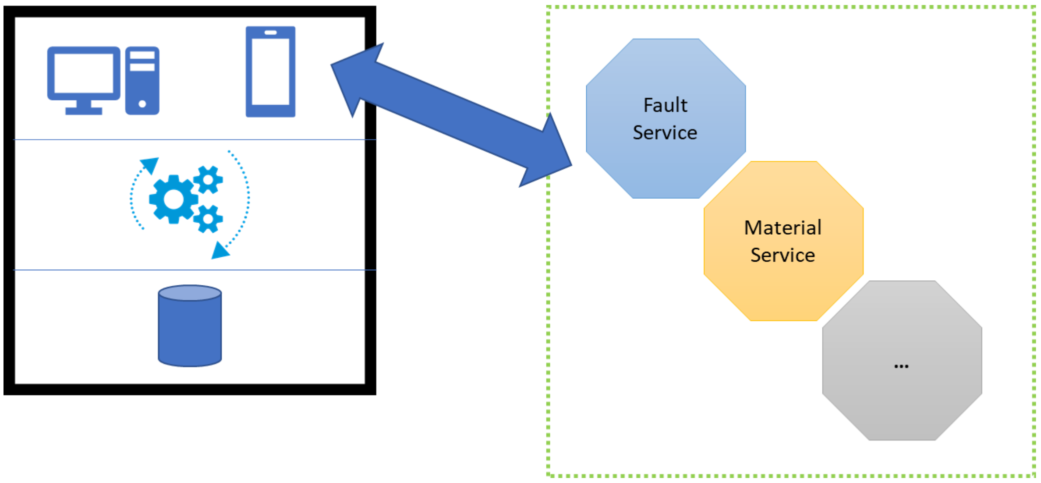


Figure 6 - Replace legacy with microservices

As mention in conversion plan 2 main challenge is to analyze the existing legacy system. Need to identify dependencies between component and process of business logic. Once microservice functionality separated it can implement without effect to monolithic core. Another challenge is to integrate microservice with monolithic core. It needs to identify usage of microservice interface, then need to remove the old way it handles it functionality. Glue code need for combine microservice API with microservice usage interface. By replacing monolithic functionality into microservice, in long run it will replace entire monolithic system to microservices.

When consider advantage of this scenario, in long run monolithic system can convert into microservices and consume advantages of microservices. Since this step done through analysis of functionality defects of process can identified and can integrate component missing with existing system.

When discuss disadvantage it is need clear view of existing monolithic system. System should analyze documents, database structure, data flow structure, client feedback. As mention it earlier it must add glue code to integrate with core system and dependency still exist with core system. Challenge of this approach is decomposition of services and integration need to be done.

It recommended to always follow microservices concepts. As mentions earlier in several time microservices should be maintain with less effort. When functionality replacement it is important to take decision on replacing functionalities put into single microservice or put into different new microservice. This decision should take based on business model aspect rather than existing system structure.

In this chapter it described the concepts of three monolithic architecture based legacy system convert into microservices based system. Three process introduce three different scenario which real system status. Single growth legacy system with more complexity with less changes, legacy system functional as expected but there are several numbers of change request to system are scenarios. Conversion can be perform using single conversion process or combination of three conversion process.

4. Implementation

4.1. Application of General Conversion Process

First, look at the high level architecture of the monolithic system which is going to convert into the microservices architecture-based system. The purpose of the system is to create communication between field officers who work to resolve defects in the phone connections. Defects information received and updated from/to middle core system not going to highlight here. Following section, we consider interfaces that field staff retrieve, monitor, resolve defects and manage their daily workflow. Follow diagram is represent components.

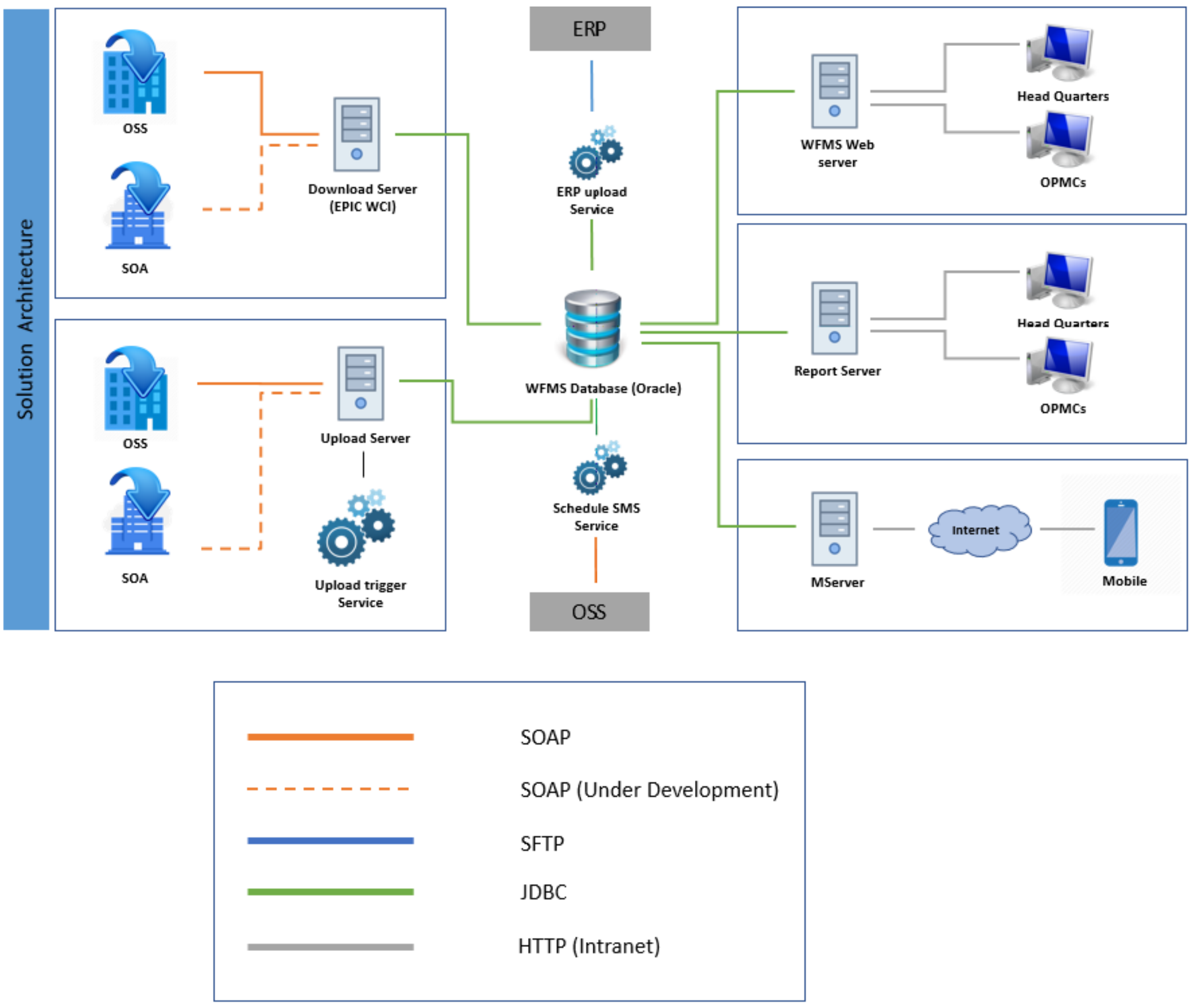


Figure 7 - High Level Diagram of Legacy System

When it considers the reason for move into microservice architecture, different reason exists. As mentioned earlier WFM system become an important system. Therefore SLT needs to evolve this system with several other systems and expand the functionality with other systems. Frequent changes regularly pop up due to the extent of the system. As mentioned in the earlier WFM system is a monolithic system since 2005 (turn into a web application in 2013) which complete for more than 10 years. Within this period different components added to the system and several modifications added by many programmers. WFM system suffers from spaghetti code and no proper documentation. From a developer's point of view, it contains messy code and the best way to apply the solution is to re-invent functionality is the best way. Customer expectation is to add new features and modules in as soon as possible to functional to improve their service to customers. Anyhow, with system status it difficult to deliver modification on considerable time.

With the above facts discussed, it decided to go for microservices architecture. Anyhow the challenge was to do the refactoring without effect to current legacy workflow, need to come up solution system within the given time range. In above discussed 3 different approaches. When considering functional replacement with microservices, the condition needs to undergo this is the best approach for refactoring/conversion to microservices.

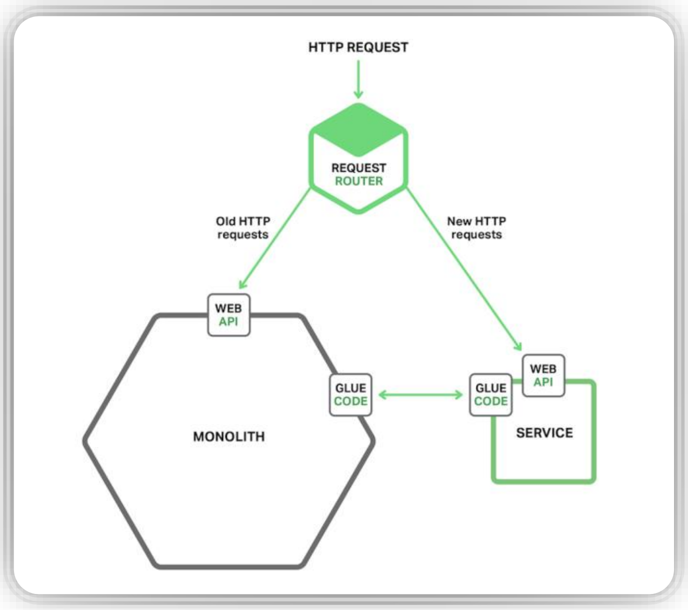


Figure 8 - Functional replace to microservices

If each function breaks down into separate microservices it will create a complex mess of microservices. Therefore it needs to consider beforehand create a new microservice for functionality. The recommended way is to follow microservices concepts. Which microservice should cover up a single business domain. By considering the business domain it will reduce the number of microservices created for function by grouping similar domain functions into single microservice.

In the real-world application, it converted functionality called “Assign/Auto Assign” which included in the legacy system. First, it analyzes the interface which communicates with the user, data pass-through communication, internal process, and dependent other functionality. Since it was only limited to single function analyze, design and implementation doesn’t take time.

Develop from scratch is another way of conversion into the microservices system. Before break into design and implementation, it should familiar with the existing legacy system. Therefore knowledge of the entire system required for the design. With the knowledge of the existing system, it should plan microservices in a way that enables loose coupling and higher cohesion.

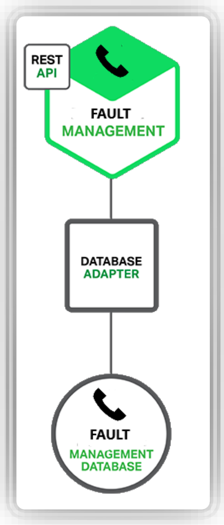


Figure 9 - Fault Microservice

Anyhow this approach not suitable for the above constraint. Because it takes considerable time to analyze, document and design entire microservices at once.

Recommending way to deal with the problematic situation is to functional wise conversion and then do the conversion to the entire application. Functional wise conversion help to gain knowledge about the existing legacy system. By acquiring knowledge about the existing system can evaluate the created microservices and its functionality. For example, “Assign/Auto Assign” functional wise microservice developed but with further function conversion to microservice it notices that it should create “Fault Management” microservice and responsible for task related to fault. All function should refer to “Fault Management” microservice.

5. Testing & Evaluation

As mention in objectives, it is the main objective to find about the pattern/processes of converting monolithic architecture to microservices architecture. Now it’s time for each methodology of conversion can perform in common monolithic e-commerce application to decomposition of microservices.

The following point will describe which are the properties that use to evaluate the result of the thesis. Based on the following properties will evaluate each conversion plan compare with each other.

5.1. Monolithic to Microservice Conversion Progress

As mention in the methodology section, it can think of different ways of conversion method. Although it won’t be useful it cannot decompose system to microservices. This property is the ultimate point that this research is can success (legacy system completely abounded by using microservice) or failure (not able to decomposition into microservices). The provided scenario should apply to common monolithic applications. Even service generated throughout this conversion plan should evaluate that fulfill microservice architecture concepts. Otherwise, it cannot get the use of advantage to contain microservice itself. Some of the important aspects of microservices need to follow which need to handle business domain-specific tiny processes (easy for manage), use DevOps to do all deployment and other configuration (error less deployment, delivery on time) and follow agile development strategies to come up solution as customer expectation.

5.2. Resources Utilization for Decomposition

Here “resource” meantime, cost and human resource allocated for the conversion process. Organizations will interest in this point since they can use their further development in less effort and cost. Even microservices are small team need to analyze huge complex monolithic system to perform. The complexity of the software system can affect the following property and therefor it will consider the same scenario for all procedures to conversion to get a considerable result. Consider the worst-case scenario that there is no proper documents, to refactor such a system get overhead time for analyze system. In this scenario software engineers along with business, analytics need to document current system functionality, behavior, and defects in current flow. Afterward, it needs time to design into a new microservice and start the implementation. This thesis, considers the worst-case scenario since WFM system haven't maintained proper documentation. It depends on the scenario going to apply.

5.3. Conversion Period

It is an important factor, that time used for conversion. Here it discusses conversion plan take for complete decomposition to microservices. Based on this factor it can decide the popularity of adopting a conversion plan. The conversion period depends on the above factor which resource utilization. Depends on works in hand time for complete products vary. As mention in the above section worst-case scenario, it takes considerable time but the progress cannot be measured (Analyze existing system and document). Conversion period relation to delivery time which is an important factor to customer and user of the system.

5.4. Comparison conversion plan

Following table evaluate conversion plan based on the above discussed factors. Note that every conversion plan estimates their value based on the same monolithic legacy system.

Property	Conversion Plan 1: Develop From Scratch	Conversion Plan 2: Adopt Microservices for New Functionality	Conversion Plan 2: Replace Functionality with Microservices
Conversion Progress	Completely convert into microservice in long run.	Partially convert into microservices.	Completely convert into microservice in long run.
Resources Utilization	Higher. (700 Man days for SLT entire module decomposition)	Based on feature/new requirement (below 50-man days as average)	10 to 20-man days.
Conversion Period	Higher. (More than 1 year for SLT module decomposition)	Based on feature/new requirement (3 Weeks)	10 Days in average.

For capture above information it used SLT real world scenario and evidence for value will be discuss more in appendices.

5.5.Conclusion

5.5.1. Summary of application

It has taken two attempts to refactor SLT WFMS legacy system to microservice based system. Although application successfully refactored at second attempt. At first attempt, tried to implement microservices system from scratch. Therefor it needed to analyze entire legacy system. Based on analyzation tried to design services that satisfy microservices concepts. Depends on facts which was legacy system complexity and dependencies it took considerable amount of time to complete. As mention in testing and evolution chapter it takes around 2 to 3 years (from project plan) to complete the task. Frequent changes and functionality expand requirement difficult for continue refactoring process. In second attempt, tried to refactor selected functionality and address new changes append to new microservice system. From the experience of attempt one it cleared that refactor should follow agile development. Replace functionality with microservices process was more suitable for refactor. Based on customer frequent usage, identified most important functionality available on legacy system. Refactoring flow designed based on customer usage functionality, dependencies with other module (“fault module”, “inventory module”) and duration to complete and deploy functionality. Functionality wise refactor took place and it combined with front end of system. While legacy system refactors, had to attended new system requirements which expand system functionality. New development doesn’t append to legacy system instead it took separate microservices development. From scratch decomposition hold due to time constrain.

SLT WFMS system refactor outcome is separate microservice system which handle most of the functionality from backend. In addition to the microservice system, derived system documentation helped to understand purpose of the service, inputs, data entities and dataflow through system.

5.5.2. Challengers and limitations

Microservice also has drawbacks its own. Complexity of microservices increases respect to the service count hold by system. Number of microservices, derived from three process different. From scratch decomposition process (refactoring process 1) derived manageable number of services. From the design plan it derived ten services that handle basic functionality of SLT WFMS. But in incremental refactor process (refactoring process 3) and new microservices for new requirements process (refactoring process 2) derived several numbers of additional services. Main challenge is to monitor and trace the operation of services. If the number of microservices increases, it difficult to analyze system failure since it need to trace variety of logs and consoles. Another challenge of microservices is communication protocols within microservices. Recommended protocols for microservices is REST. REST protocols difficult to enforce security requirements (even adding security mechanism to REST, reason to slow down communication between services). Microservice main concern is high reliability. It needs to manage fault tolerance mechanism. Failure to response for request need to design before development stage. Discover and response time from microservices is important fact. Therefor it needs to adjust protocol or waiting time at client side for systems functionality. If the microservice repeatedly need to change or rework it may need to split into more services. Since microservice handle small business concern it had not changed afterwards. These challengers depend on situation and thinking pattern. Following summarization of challengers and limitation of research.

- Higher number of microservices reason to increase system complexity.
- Need to arrange proper communication protocol between microservices.
- Enforce security mechanism to solution.
- Monitor and trace microservices behavior.

5.5.3. Future works

For the future work it not discussed apply non-functional requirement adhere along with decomposition. For example, non-functional requirements such as security and trace important when it come for real time functions such as bank transaction scenarios. Proper documentation easier the maintains of any system. Since microservice handle small business concern it can suggest proper documentation template. Document template reason improve readability, enhancement, appoint development milestone, trace the development progress and most importantly understand system in less time. Several tools available for easier to manage and develop microservices. For example, service discovery which manage service registry and help to other service to identify available service, central configuration service which hold entire configuration within separate service (it easier to modify configuration information without effect to operation of microservices) and clustering service to deploy microservices in server (clustering service enable automated deployment, version control and fault recovery). Application of DevOps is separate area of microservice which automate deployment, fault tolerance and load balancing. Following are some suggested area and technology for enhance research area.

Domain for Enhancement	Suggested Technologies
Nonfunctional requirements	Secure communication protocols. (SSL, encryption, Digital Sign)
System documentation	Usage of documentation template.
Monitor and tracing	Central configuration, Monitoring service (Docker, apache application manager)
Fault tolerance and automation	Service clusters, load balancing (Kubernetes, Kong)

In further microservice domain can further expand with latest technology which currently not available at write of this research.

6. References

[1] M. Samek, "Patterns of Thinking in Software Development," BARR Group, 04 May 2016. [Online]. Available: https://barrgroup.com/Embedded-Systems/How-To/Patterns-of-Thinking-in-Software-Development.

[2] B. PUTANO, "Most Popular and Influential Programming Languages of 2018," 18 December 2017. [Online]. Available: https://stackify.com/popular-programming-languages-2018/.

[3] B. G. D. K. T. J. R. T. Wade L.Schulz, "Evaluation of relational and NoSQL database architectures to manage genomic annotations," United States, 2016.

[4] P. Wayner, "The top 5 software architecture patterns: How to make the right choice," [Online].

[5] J. B. A. Z. S. W. Jonas Fritzsch, "From Monolith to Microservices: A Classification of Refactoring Approaches," Cornell University, 2018.

[6] C. Cancialosi, "Outdated Tech Is Costing You More Than You Think," 16 August 2017. [Online]. Available: https://www.forbes.com/sites/chriscancialosi/2017/08/16/outdated-tech-is-costing-you-more-than-you-think/#cf1a1c910994.

[7] R. Sakhuja, "5 Reasons why Microservices have become so popular in the last 2 years," 12 March 2016. [Online]. Available: https://www.linkedin.com/pulse/5-reasons-why-microservices-have-become-so-popular-last-sakhuja.

[8] I. Miri, "Microservices vs SOA," DZone, 01 Jun 2016. [Online]. Available: https://dzone.com/articles/microservices-vs-soa-2.

[9] W. H. Holger Knoche, "Using Microservices for Legacy Software Modernization," Research Gate, 2018.

[10] J. Lee, "All About Microservices," 24 October 2017. [Online]. Available: https://dzone.com/articles/all-about-microservices.

[11] S. u. Haq, "Introduction to Monolithic Architecture and MicroServices Architecture," 2 May 2018. [Online]. Available: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

A. Appendix

Microservices can develop using various tools currently available follow discuss some tools that can be used to develop microservices. Follow section is limited to Java programming language and there is vast number of tools that related to other languages.

i. Config Servers

To keep the Properties file centralized and shared by all Microservices, we will create a config server which is itself a Microservice and manages all microservices properties files and those files are versioned controlled; any change in the properties will automatically publish to all microservices without restarting the services. One thing to remember is that every microservice communicates with the config server to get properties values, so the config server must be a highly available component; if it fails, then all microservices fail because it can't resolute the properties values! So, we should take care of the scenario - the config server should not be an SPF (single point of failure), so we will spin up more than one container for the config server.

ii. Discovery Server

The main goal of Microservices is decentralizing the different components based on the business features, so that each component - aka microservice - can be scaled as per need, so for a microservice, there are multiple instances and we can add and remove instances as per the need, so the way monoliths do load balancing is not going to work in a microservice paradigm. As it spawns containers on the fly, containers have dynamic IP addresses, so to track all instances of a service, a manager service will be needed, so when the containers are spawned, it registers itself to the manager and the manager keeps the track of the instances; if a service is removed, the manager removes it from the manager's service registry. If other services need to communicate with each other, it contacts a discovery service to get the instance of another service. Again, this is a highly available component; if the discovery service is down, microservices can't communicate with each other, so the discovery service must have multiple instances.

iii. Gateway Service

A microservice is a collection of independent services which collectively produces a business functionality. Every microservice publishes an API, generally a REST API, so as a client, it is cumbersome to manage so many endpoint URLs to communicate with. Also, think about another perspective: if some application wants to build an authentication framework or security checking, they must implement across all services, so that would be repeating itself against DRY. If we have a Gateway service, which is internet facing, the client will call only one endpoint and it delegates the call to an actual microservice, and all the authentication or security checking will be done in the gateway service.

iv. Concerns in Legacy Systems

To grasp the technology that pop out with time need better understanding of domain area. For example, Microservices are architectural design pattern it can be used for places where need high scalability, maintenance and service-based implementation. Anyhow, to adopt with the new technology need to familiar with concepts, tools and strategies. As mention earlier microservices is new to domain. To adopt with microservices need to understand basic concept of microservice such as focus on CI/CD concepts. Someone can point that acquiring and familiar with those new technologies responsible for senior engineers or innovation team in the company. However, everyone responsibility to keep up to date familiar with newly technology and concepts. Limited to pattern of development process reason for many pitfalls. Organization should motivate team to improve knowledge with in organization.

Most of the legacy systems are take monolithic based architecture. These systems may develop using outdated tools or newest tools. Most of the time team allocate for product(s) to maintain and improve. Some of the team may responsible for maintaining legacy system. This thesis not mentioning that legacy systems are pitfalls for the organization. Although with time being it need to modify, change or even refactor based on situation. In practical situation development team try to add patch or append existing functionality with new required functionality/modification to system. Its reason for code base growth and difficult for maintains. Legacy systems are main barrier for company to move forward with new changes.

Some of the concerns are Difficult for maintain and test with growth of code base, Difficult for document reason for complex document or no document related to system, Considerable resources allocation such as team, budget or time, constrain into technology stack, difficult or no way to adopt new technology stack such as frameworks.

Most organization fear to invest on innovating organization process, consider software development process and for new production. Most of the company even though it invests failed to continue by judging short term outcome. Most of the organization fear for change because of failure reason for software crisis. Experienced employees moving out of the organization is another fallback of organization. This thesis not much concern about this phenomenon.

v. Continuous Integration with Version Control

There is a working monolithic software system, and the team responsible for maintaining the system has decided to migrate this system to microservices. The first step toward continuous delivery is to set up continuous integration. Continuous integration automates the build and test processes and ensures the availability of production-ready artifacts. Normally, a continuous integration pipeline contains a code repository, an artifact repository, and a continuous integration server. First, each service should be placed in a separate repository, which enables a clearer history and separates the build life cycle of each service. Then, a continuous integration job should be created for each service. Each time a service's code repository changes, the job should be triggered. The job's responsibility includes fetching the new code from the repository, running the tests against the new code, building the corresponding artifacts, and pushing these artifacts to the artifact repository. Failure to execute each of these steps should terminate the job from proceeding and informs the corresponding development team of the occurred errors. This team should not do anything else until addressing the reported errors. One simple rule in continuous integration is that new changes should not break the system's stability and should pass all the predefined tests.

vi. Load Balancer

A software system has been decomposed to a set of small services to use a microservices architectural style, and a service registry has been set up. In the production environment, numerous instances of each service exist. Each service can be a client of the rest of the services in the system. Each service, as a client, should have an internal load balancer, which fetches the list of available instances of a desired service from the service registry, eg, through a service registry client. Then, this internal load balancer can balance the load between the available instances using local metrics, e.g., the response time of the instances. An internal load balancer removes the burden of setting up an external load balancer and brings in the possibility of having different load balancing mechanisms in different clients of a service.

vii. Usage of Circuit Breaker Pattern

A software system has been decomposed to a set of small services to use a microservices architectural style. Some of the end user requests need interservice communications in the internal system. The service consumer can use a circuit breaker when calling the service provider. When a service provider is available, this component would not do anything (the close circuit state). It monitors the recent responses from the service provider and will act appropriately when the number of failure responses passes a predefined threshold (the open circuit state). The corresponding action could be either returning a meaningful response code or returning the latest cached data from the service provider (if it is acceptable for that specific response). After a specific timeout, in order to check the service provider's availability, the component will try to access the service provider again (the half-open circuit state), and when there is a successful attempt, the state will be changed to a close circuit state. Otherwise, the state will be modified to an open circuit state.

viii. Containerize the Services

A software system has been decomposed to a set of small services to use a microservices architectural style, and a continuous integration pipeline is in place and is working. Each service needs a specific environment to run correctly, which is either set up manually or through a configuration management tool. The differences between the development and production environments can cause some problems, e.g., the same code may produce different behaviors in these two environments. Therefore, the deployment of services in the production environment becomes a cumbersome task. As each service may need a different environment for its deployment, a solution could be the deployment of each service in a virtual machine in isolation with their own desired environment using configuration management tools. The downside is that due to virtualization, a lot of resources are wasted for service isolation, and configuration. management is another layer of complexity in the deployment. Compared with the virtualization, containerization is more lightweight, and it can remove the need for configuration management tools since there are a lot of ready images in the central repositories containing different applications, and any further configuration can be done in the new images building stage. For this solution, add another step to the continuous integration pipeline to build container images and store the images in a private image repository. These images can then be run in both production and development environments that produce the same behavior. Each service should have its own container image creation configuration; the script for running any other required services' containers should reside inside its code repository. It is a good practice to add environment variables as a high priority source for populating the software configuration. In this way, the configuration keys that can have different values in different environments, eg, database URL or any credentials, and they can be injected easily in the container creation phase. Having a list of required environment variables for running a service in its code repository is a good practice as it makes stakeholders aware of these changing variables.

ix. Monolithic and Microservice Architecture

a. Monolithic Architecture

[9]Monolith architecture is an old but even currently active software architecture that contain large applications with big monolithic codebase. Main key point of monolithic architecture is single code base. Since project handle by single code base it uses to deploy whole system once. Continues delivery cannot achieve with monolithic architecture, when whole system needs to update. Later time for avoid complexity from monolithic architecture it tries to use layers. From its high-level structure distributed into vertical 3 or more layers. Monolith way of development of software is common way of application development. At begin it is easier, single code base is simple to design, develop and deployment. Most applications can be simple in the beginning, but as the application grows, so does its complexity. A typical way to handle complexity in an application that has a monolithic architecture is to split the application into different layers.

Continues evolvement in technology and related domain reason for innovation take place. Monolith architecture take follow architecture using single code base.

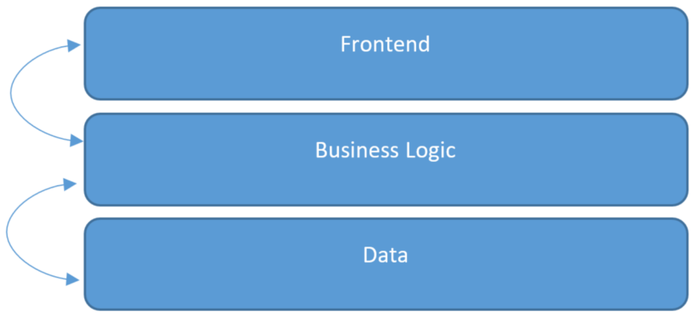


Figure 10 - High level diagram for Monolith Architecture

b. Microservices Architecture

[10]Microservices is an architecture which mainly focus on loosely couple services architecture style. These services expect to fulfill business capabilities of organization. [11]These services are small (smaller code base) and focus on given manageable responsibilities. Organization process build using tiny several services which interact each other using API of each services.

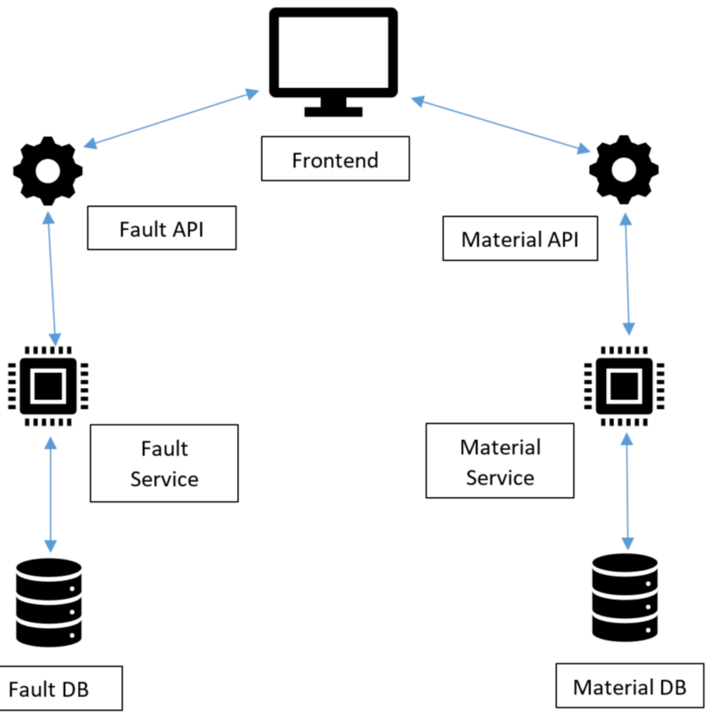


Figure 11 - High level diagram for Microservice Architecture

Microservices are provide higher modularity by its architectural structure. Since each service independent easy for develop, maintain and deploy. Service API is responsible for expose microservice which enable clear boundaries between services. Single Responsible Principle enable by its nature by modularity and small tiny functionalities.

By the way, to complete business task it may take more than one service call. Service call may reason for communication delay. Communication delay is one of drawback come along with microservices. Therefor identify microservices before conversion is tedious task. Microservice can achieve many advantages reason to many valuable properties such as modularity, Single Responsible property, small etc.

When developing microservices, the goal is to have services that are loosely coupled and highly cohesive. Loose coupling means that services should not know anything about the internals of other services. This is achieved with microservices as they have clear boundaries by nature and only communicate through interfaces that each microservice publishes. Cohesion can be described as the tightness of related features in different modules. When microservices are separated correctly they adhere to SRP and they have a single business context on which they operate. If these qualities are applied to microservices, it means that the microservices are also highly cohesive. In order to achieve loose coupling with microservice architecture, it might make sense to duplicate some of the code. Typically, developers have been taught to follow the DRY (do not repeat yourself) principle. DRY states that the same code should not be repeated in the codebase but instead the code should be reused. This is good advice inside one microservices but when multiple microservices share same code problems can occur. If one service requires a change to the shared code it means that all the services which use the same shared library must be also updated and deployed. This means that the services are now tightly coupled. The situation can be solved by rather duplicating the code. It gives the freedom for each of the services to be independent.