International Journal of Scientific Research and Engineering Development-– Volume 4 Issue 3, May -June 2021
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Understanding Migration from Monolithic to Microservice Architecture and its Challenges

Abijith Trichur Ramachandran, B.E[1,2], Abhishek R, B.E[1], Mamatha G.S, M.Tech, Ph.D[1], Rashmi R, B.E(CSE), M.Tech(SE), Ph.D[1], Badrinath K, B.E (E&C), M.E (Applied Electronics), Ph.D (C&IS)[1], Manojkumar Parmar, B.Tech(ECE), M.Sc(Innovation & Entrepreneurship)[2,3]

[1]Department of Computer Science and Engineering, RV College of Engineering, Mysore Rd, RV Vidyaniketan, Post, Bengaluru, Karnataka 560059, India
[2]Robert Bosch Engineering and Business Solutions Private Limited, Bengaluru, India
[3]HEC Paris, Jouy-en-Josas Cedex, France

Abstract:

Cloud computing has already become a core sector for the enterprises in the current digital transformation. It's of prime importance for companies to choose suitable architectures and migration techniques to gain the benefits of the cloud. Microservice architecture blends well with cloud computing and turns out to be a de facto standard for all enterprise level applications. Most of the applications in the early stages start as a monolithic application. So, companies should migrate monoliths into microservices as the complexity increases. This is a difficult task. This paper discusses the challenges of migration, pre-assessment needed for migration and the various methods for migration.

Keywords: Cloud computing, monolithic architecture, microservice architecture, migration of cloud service, containerization, microservices identification, cloud architecture scalability, migration challenges, DevOps

I. INTRODUCTION

Microservices have grown in popularity in recent years as DevOps approaches and container technologies like Kubernetes and Docker have spread. Since 2014, there has been a substantial growth from the significant use of microservice style of architecture, as seen in the service based softwares where the microservice based architecture has been much preferred to other software design frameworks. Microservices are self-contained modules that separate well defined business functions. A microservice also operates on its own process and interacts with other microservices via simplified interfaces and lightweight protocols [1]. Microservices architecture (MSA) is built on a share-nothing concept and is based on years of system engineering and application engineering experience. The whole architectural style solely focuses on organizing a system into a group of loosely connected sets of small resources that are separated into small, cohesive, and self-contained groups that can communicate with each other. Many businesses and organisations, including Netflix, Amazon, and eBay, Guardian have moved their software and systems to the cloud because of the cloud infrastructure paradigm which enables them to scale up or down their processing requirements according to their usage and improve theircontinuous delivery of massive and complex softwares while maintaining stability and versatility of the application framework. MSA architecture criteria are being used to convert systems of conventional architectural styles into MSA [2]. MSA's services communicate with one another using lightweight protocols like HTTP resources such as an API, where each and every service runs in its own dedicated process. Monolithic architecture, on the contrary, is a multi-service application with a common code base. These programs interact with external networks or users via a variety of interfaces, including Web apps, HTML websites, and REST APIs [3] .

Organizational resilience, or the ability to rapidly adapt products and services in response to changes in the business environment, has become extremely important for maintaining long-term competitive and strategic advantage. Not only in the technology industry, but around the platform, information and communication technologies (ICT) are crucialinstigators of enhancing business innovation (e.g., Uber and Airbnb). Massive monolithic software structures, such as Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM), influence many corporate capacities and

 ©IJSRED: All Rights are Reserved

operations. A monolith is described as a set of software programs that cannot be computed independently ofone another. As processes evolve and become even more sophisticated, changing them becomes morecomplicated,

therefore can inhibit an organization's potential to react quickly to modifications in the external environment, which includes changes in consumer and contender behaviour. [4].

Architecting microservices can be approached from a variety of angles. Aside from engineering and operational dimensions, the designing of these services has now become a prime focus for research. For instance, there are studies that focus only on determining the appropriate scale, granularity, and quantity of microservices. This is a very hard process because design choices are influenced by many factors such as harmony, coupling, and consistency characteristics. There's also the problem of runtime instability. Having too many microservices will lead to severe coordination complexity, whereas microservices with too few services will maintain the pitfalls of monoliths. The method of domain-driven architecture is often discussed when it comes to designing microservices. However, there are other options that are based on parameters, algorithms, and assessment techniques that significantly vary between approaches. It is often difficult to pick a suitable strategy when there are so many different methods that are viable to be implemented [5].

In this work we will primarily focus on understanding the difference and implementation of monolithic and MSA. We also focus on the possibilities in the migration of monolithic to a microservice based architecture to overcome severe limitations proposed in monolithic architecture.

II. MONOLITHIC ARCHITECTURE

Services are built on top of a single platform and that code base is shared between the developers. Whenever a developer makes any changes to the code base he should make sure that other services in that same code base are operational. Adding up more services introduces more complexity which restricts the company's ability to experiment with new versions and functionality. Whenever new versions are introduced all services are being restarted and this results in poor user experience. There is also the problem of single point failure that if a single service is crashed this brings down all services. [9]Monolithic systems cost more with more complexity [6].

Case Study: **Monolithic architecture in cloud computing**: We already know that monolithic applications have a single codebase. Hence these applications are to be built using MVC framework. MVC stands for Model View Controller.. Developer has only two REST services running, these REST services are being utilized by MVC to run the web application. This type of system is limited to only one server environment, limiting the server resources and making it non scalable and limiting the number of users and high load. It can also be deployed in multiple server environments [10].

Monolithic architecture in space systems: Space systems have to deal with a number of uncertain situations for their life span. This is a challenging task for developers to code the different services and maintain it in a single code base. Technology also changes day by day this makes the designed system obsolete and less efficient. Various design evaluations also need to be carried out for different uncertain situations. Designing for uncertain situations increases the cost and results in growth of complexity [9].

Advantages of monolithic architecture:

- It's very easy to get started with development and hence, most new projects start out as a monolithic application
- It's easy to conduct unit tests and integration tests and hence suits test driven development
- Deployment is easy. Hence we can use this architecture if the application is of small to medium size
- It supports horizontal scalability through replication and a load balancer

Disadvantages of monolithic architecture:

- Complexity of the system grows as services are being added.
- Experimenting ability and adding new versions and functionality is restricted.
- Services should be restarted after adding a new service which may result in poor user experience.
- Whenever a single service crashes resulting in the whole system crash. Code base is limited to one.
- It's a burden to developers as they have to maintain full functionality after adding a new service.

Hence, whenever a new project is started, one can begin with a monolithic architecture and as the applications grows and evolves, the business requirements and functionality has to be analysed and assess whether or not migration to microservices will be useful. Generally, migration happens only when the system becomes complex and becomes difficult to manage [6].

III. MICROSERVICE ARCHITECTURE

 ©IJSRED: All Rights are Reserved

MSA is an approach to architect a software project using loosely coupled modules or services. Services are designed based on core features of the project. Since the services are independent of each other, they can be scaled and deployed simultaneously and separately, thereby increasing productivity. Microservice architecture can be viewed as an upgrade over Service-Oriented Architecture or N-layered architecture. Services in Service-oriented architecture are dependent on each other to some extent as services can share the same database or one service can depend on the other [7]. HTTP/REST along with JSON is the most popular architectural design paradigm for microservice implementation [3]. Hence, a micro service system comprises multiple collaborating services, generally without a central control [8].

Advantages of micro services:

- Microservice is independent of programming language and technology stack [21]. Hence each service can have its own technology stack for implementing it's functionality
- No single point of failure. This resilience [8] feature its very important while building microservices
- Scaling the application becomes very easy as the services can be selectively scaled based on requirement [21], whereas in monolith entire application has to horizontally replicated for increasing scalability, thereby incurring more cost
- Microservices can be deployed quickly and with ease. It suits applications which evolve over time. It's an important feature for continuous integration and continuous deployment (CI/CD)
- It increases productivity of the organization as different teams can work on different services simultaneously as they are nearly independent of each other
- Microservice is the key to DevOps, which focuses on automation and CI/CD and improves collaboration between teams [22] and helps in conducting performance benchmarks [23]
- Microservices blend very well with containerization technology like docker [24]. Containers help in faster and easy deployment of the microservice

Implementation/Deployment methods for microservices:

- Deployment of each microservice in a separate Virtual Machine(VM)
- Deploying microservices in containers like docker
- By utilizing Kubernetes, which is a container orchestration service

The paper [24] concludes that the response time is a higher priority than deploying microservices in VM. It is suitable as it performs better by a factor of 1.25 with respect to containerization in AWS cloud setup. If your priority is faster deployment, then we should use conatinersation as its lightweight and flexible and causes very minimal impact on processing, network and memory [25]

IV. THE ASSESSMENT OF MICROSERVICE FRAMEWORK

Aim of the assessment framework is to assist businesses in debating benefits of relocation and making decisions on real time data and the problems in their current monolithic frameworks. This framework is not focused on advising a particular judgement for example migration based on a metric rather to advise different organisations in not overlooking crucial details and finding reasons on the most comprehensive collection of data available when determining whether to migrate or not to migrate. This results the system must be adapted to individual circumstances of the organization that uses it. Below are the four steps in the framework

Identifying motivational factors: Before migration from monolithic architecture to microservices architecture companies must have the clarification for the migration. Usually companies migrate for solutions to some issues that are solved differently. Migration may impact negatively. Companies should keep in mind that migration may create some issues. Companies still want to migrate so they can go on and collect metrics.

Identification of metrics: In order to migrate the companies should first perform analysis on their existing monolithic architecture system. Companies don't have to consider only one metric for analysis. Better they consider more metrics. Sub characteristics of metrics also should not be ignored. Some of the metrics are performance efficiency, reliability, maintainability, cost etc.

Decision about migration: After discussion about the result of metrics being collected, the company may think of is it necessary to migrate or not. Low performance may be because of improper implementation of algorithms in such cases companies may think of implementing algorithms in an efficient manner.

Actual Migration: Teams working in a company start migrating from monolithic architecture to microservices architecture. Recommended way of migration is automating the process of collectingvarious results for metrics and setting up measurement tools [7].

V. MSA BOUNDARY DESIGN PRINCIPLES

 ©IJSRED: All Rights are Reserved

Do one thing exceptionally well: Microservices can have a high level of cohesion between them because they enclose items both methods and data that should be used together. Micro service is intended to be one stop point to get, add or modify items not bigger than a team: Every micro service must be big such that it can be easily handled by a team. Single team can easily manage a micro service. Context of a micro service should be single such that context can be easily understood by a person. Ideally, there should be no more than a fewhundred lines of code. Micro services can also contain zero code if a GUI based designer IDE is used. Micro services which are smaller can be easily refactored.

Organizing data in groups: Functional boundary is based on data that the microservice has, operations performed by the microservice, views provided by the microservice for the data. Data required for a single API call frequently belongs to a single microservice. If keeping data combined makes the microservice simple that is good. In the same way keeping separate data does not affect adversely then this is also good.

Data items should not be shared in a bad manner: There is a need for a proper data item sharing mechanism. Proper API calls or event based interaction needs to be implemented.

Few tables or less data store: Each micro service should only contain a few tables as this may result in a big database. Microservices should optimize the data storing mechanism for example relational database, key value pair based storing.

Independent selection of technology: Microservices offer flexibility in selecting technology. Often there may be requirement changes or any constraints may lead to change in technology.

Independent release: Micro services should be coupled loosely. Microservices can be easily modified and re-deployed without affecting any other microservices.

Microservices with a lot of interaction should be kept to a minimum: Atomic microservices having interdependence should be removed. Microservices with lots of interaction should be merged and to be kept as a single microservice [21].

VI. MONOLITH APPLICATION TO MICROSERVICES MIGRATION TECHNIQUES

1. General methodology for decomposition of monolithic application to microservices:

Task-1:Decoupling Monolith Application

Through the addition of an intermediate facade layer between the frontend layer and the backend layer decoupling can be achieved. Every facade should act as a pass-through layer involving no transformation of data, thereby mimicking the monolithic layer. For implementing any additional user interfaces (channels), each facade layer can be further divided into a service implementing the specific microservice (if identified). Service mediation layer shown in Figure 1 can be brought in above service or facade layer for providing runtime control to channel-service mapping [14].

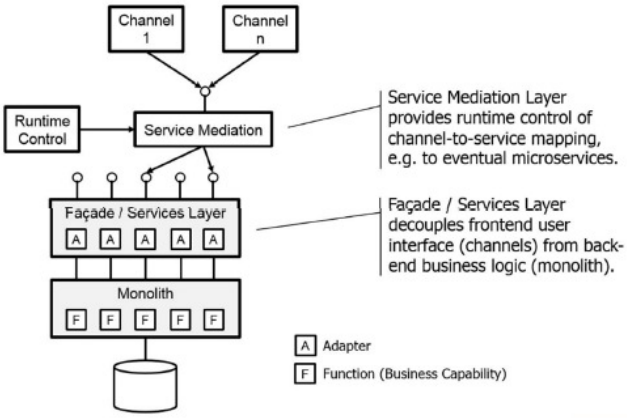


Figure 1: Refactoring user-interface from monolith [14]

Task-2:Develop local microservices

According to [14] business functionalities play an important role in migration from monolithic to microservice architecture. The authors in [15] propose a strategy revolving around business functionality for microservice identification through a flow chart shown in Figure 2..

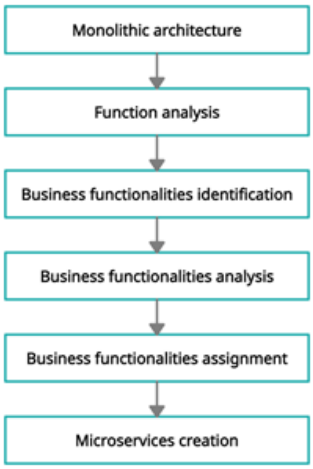


Figure 2: Flow chart for decomposition of monolith into microservices [15]

Step-1) Functional analysis: Analyse the functions in the monolithic application based on its usage frequency and its LOC/size. After the analysis, we can make decisions about

 ©IJSRED: All Rights are Reserved

splitting the function into smaller ones or whether we should aggregate multiple functions into a single function.

Step-2) Business functionalities identification: There will be specific business functionalities associated with the monolithic application. Monolithic developers can guide in identifying such functionalities. For complex applications, in [19] the author describes a machine learning technique with the required alterations for identifying the functionalities.

Step-3) Business functionalities analysis: Business functionalities are analysed to obtain the usage rate and other useful statistics. This can be achieved through the simulation of the entire system using machine learning or human interaction.

Step-4) Business functionalities assignment: From the statistics information, granularity and quality of microservice can be improved. If a specific functionality is used more often as compared to others, then a separate microservice should be dedicated to it. If few functionalities have similar scope and usage patterns, then they can be grouped into a single microservice. This is one approach of mapping business functionalities to microservices based on usage pattern. Similar mappings can be drawn from other statistics information.

Step-5) Microservices creation: Based on the allocation of business functionalities to microservices from the previous step, microservice is created by the development of functions related to business functionalities within it.

We must ensure that the required amount of cohesion & loose coupling is maintained for every microservice. Also, the corelogic and data structure in the function in monolith and microservice should match.

Tas-3:Local Microservices implementation: After the microservice is developed and unit tested, it should be deployed in a local set-up. After successful local deployment, the channel-service mapping can be modified separately. The following are followed for each microservice locally:

step-1) Data migration from monolith to microservice

step-2) Ensure that the output of the created microservice and the corresponding monolithic feature is consistent by parallely running both microservice and monolith through the channel

step-3) Map the channel to invoke microservice for the corresponding monolithic feature

The above process can be repeated til all channels invoke microservices-only functions. Service mediation technique helps us to switch between monolith and microservice easily during migration without much modification.

Task-4: Deploy Microservices to cloud

API gateway creates a single point of access for the microservices. This API gateway can be implemented in the required cloud environment. After setting up the API gateway, microservices can be deployed in the cloud. Also, required tools for monitoring and management of microservices can also be deployed. The Figure 3 shows a typical microservice architecture deployed in AWS.

Figure 3: Sample microservice architecture in AWS

Task-5: Deploy Microservices to cloud

The steps specified in the local implementation of microservices in phase-3 is repeated with cloud version of the microservices to ensure the functionality is consistent

Task-6:Decommission Monolith

After phase-3, services of monolith are not utilized, hence it can be removed if the corresponding microservice is implemented successfully.

2. Storage-Based Method

The authors in [16], propose a technique consisting of four steps for identification of microservices from monolithic systems.

Step-1) Decomposition of database: The database tables have to be broken into sub-tables, such that each sub-table should map to a specific business functionality. This requires a thorough understanding of the entire business process.

Step-2) Dependency Graph: This step involves creation of a dependency graph involving facades, database tables [and sub-tables] and business functionalities. This maps the dependency between business functionality and the database.

Step-3) Database tables and facades pairs: From the dependency graph, identify the unique pairs of database tables and facade pairs and link it with business functionality. The number of unique pairs increases with the complexity of the dependency graph.

Step-4) Microservice candidates: Here, we identify the suitable candidates to be converted into microservices from the unique pairs. The code of the facade and business functionality which are on the path connecting the database table and facade has to be inspected.

3. Code-Based Method Evaluation

The authors in [17] extricated microservices from monolithic application through a couple of transformations. The steps are:

step-1) Construction step: The monolith is converted into a graphical representation, wherein each node indicates a class/functionality in the monolith and the edges indicate the node's cohesion with the other classes. Quality of code is

determined by the coupling factor of the monolith. Hence, higher the code quality, lower is the number of edges.

step-2) Clustering step: Here, the graph is sliced into separate components, each representing a suitable microservice. The authors have suggested three techniques to achieve this: logical coupling, semantic coupling and contributor coupling.

This method of extracting microservices is directly dependent on the code as if standard clean code-practices are followed, it's easier to identify microservices since classes will have specific responsibility, fewer dependencies and meaning naming.

4. Business-Domain-Based Method Evaluation

The authors in [18] propose a migration technique which is based on the Software Development Life Cycle [SDLC]. Two analytical steps are used for the migration process.

step-1) DDD (Domain-Driven Design analysis): Suitable microservice candidates in monolith are identified using DDD and bounded context analysis. DDD identifies unique domain modules in a domain and extracts all the domains in the monolithic solution. This technique helps in identifying loosely coupled microservices based on domain.

step-2) Database analysis: Here the database schema is analysed. Technically speaking, every microserice should be mapped to a unique database. This prevents cohesion between microservices. We can reverse engineer this fact and break down the large monolithic database into smaller databases, each bound to a specific functionality. Foreign keys give us a hint towards the suitable microservice candidate.The following table indicates the suitable method for monolithic decomposition based on different scenarios.

Table 1: Table representing various migration methods for different scenarios

Scenario	Storage-based	Code-based	Business domain based
Functions have higher priority	YES	YES	NO
Business domains have higher priority	NO	YES	YES
Programming language independent	YES	NO	YES
For Backend application	NO	YES	NO

VII. CHALLENGES DURING MIGRATION

Due to the growing popularity of microservices, they are increasingly suitable to be deployed in instances where the costs considerably surpass the advantages. One rationale could be that the developed project would benefit from being developed in a monolithic architecture. This isn't to say it should never be modular; it just means that the modules don't have to be as separated as microservices. Microservices cannot and will never be the best option or a solution in every situation aren't, and will never be, the best solution in every situation. However, there are scenarios in which microservices would be a perfect match but teams fail to build them successfully are more interesting and occur more often depending on the organisation. There could be a variety of reasons for this [11].

The challenges and difficulties with the implementation of a microservice architecture can be classified into two sections; i.e technical and organisational challenges. It's critical to get both of the sets of challenges in the right manner. The problems range depending on whether the programme is being built from the ground up or migrating a large monolithic codebase to microservices. We look at some of the difficulties that come with restructuring current monolithic apps into microservice based architecture. The majority of these issues must be addressed when using the MSA from the beginning. The most significant distinctions are that no major organisational changes are required, and no reworking is required; however, if teams want to engage with microservice architecture from the foundation, the choice of services and their business needs may be more difficult [13].

Technical Set of Challenges

1. Service Modularization and Refactoring

Before it is possible to use microservice architecture, a number of technological difficulties must be overcome. When a monolithic programme is used as a starting point, the organisation is quite probably already familiar with the domain and knows where modularization of the application can take place. The most difficult obstacle in these situations are separating these services. Refactoring the services out of the monolithic framework can take quite a long time and energy. As a result, the refactoring of microservices must always be done in stages. And moreover even if it may be quicker, new functions must not be attached to monoliths when constructing them. To replace the old monolithic code, enterprises should grow their microservices offering and introduce new microservices. Using this method, the organisation is gradually migrating the majority of its codebase to microservices. It's critical to use great caution when doing this refactoring, as there's a risk of developing new flaws in current functionalities [13]. Although the general

 ©IJSRED: All Rights are Reserved

difficulty will persist, two components of it may be addressed. First, usage of tools could make redesigning a microservices-based system simpler, even if there are unexpected repercussions in terms of infrastructure or development reliance. Second, a shift to asynchronous communication, allows more widespread use of libraries that incorporate stability patterns, and more advanced runtime environments could all aid in the resolution of stability problems. A totally serverless strategy, on the other hand, incorporates a lot of trust on the underlying platform and the services it provides. Again, this comes at the expense of a greater reliance on a certain environment. Which results in a direct opposition to one of the fundamental purposes of microservices [11].

2. Testing

Testing can be thought of as a facilitator for the entire refactoring process. If the majority of the tests are executed manually, it may be preferable to focus on getting the automated test coverage up and running before restructuring to microservices. Refactoring is aided by well written automatic test coverage, and it also allows you to get much more out of the migrating microservices. Only if it is easy to validate that the program accomplishes what it is required to achieve can microservices be delivered often. Based on the needs of the application, developers can use a variety of automated testing methodologies. Microservices are essentially interdependent with continuous integration and continuous delivery. It is quite difficult to manage several services, coordinating deployments, and validating the behaviour of the system without these two operations [13].

3. Integration

Integration of various microservices is one of the most difficult tasks. Since the teams may choose to utilise various programming languages while creating services, it is not advisable to attach the interaction between services to a certain platform or a technology. It is preferable to use a technology that does not demand the use of a certain programming language. When it comes to microservices integration, there are plenty of related issues which are required to be considered. The microservice interfaces should always be straightforward to use and must possess high backwards support, so that even when we add new functionalities, the clients accessing the service do not require to be modified. It must, like any good interface, encapsulate the implementation details within [13]. In microservice designs, user interfaces (UI) are a fundamental component. This is largely due to the fact that many of the approach's proponents, who are often back-end architects, haven't focused on them. As a result, systems with a monolithic frontend and a multitude of backend microservices are possible. While such systems can be just fine on various occasions, they frequently stifle the aims of microservices since all of the drawbacks of a monolithic architecture still remain. Modularization of various UI services, whether web, native, or hybrid, as well as their connection to microservices or their participating entities, is frequently required to assist enterprises in implementing a full-stack microservice infrastructure [11].

4. Service Granularity

Another challenge is the shortfall of consensus on how large microservices should be. Despite the fact that the name implies that microservices must be as compact as viable, project teams understand this precept in a variety of ways. Microservices with a few dozen lines of code (LOC) are used by some teams. Others make a microservice out of a few KLOCs by using a few dozen classes, and potentially a couple of database entities. These mid-sized services has the ability to communicate synchronously or asynchronously. The self-contained systems consists of an approach, which is likely a version of microservices, which supports embedding UIs, notably web UIs, as part of a service and promotes depending on frontend integration as much as practicably possible. Each of the approaches has its own set of merits. The fact that they're all named "microservices" indicates that there's a prospective for creating a set of patterns to aid design choices when breaking a domain into microservices and scaling each and every service.

5. Resource Monitoring and Management

The amount and diversification of infrastructure resources (such as, virtual machines, containers, thread pools, and logs) are very much required to be continuously monitored and managed at runtime as the size and complexity of microservice applications grows. Furthermore, services may be distributed over many regions and availability zones, making it even more difficult to keep track of their state and behaviour. Finally, while current monitoring systems include an expanding level of automation, application developers may find themselves inundated with monitoring events thus make them unable to maintain timely strategic decisions. The ability to create adequate alert thresholds and limits so that developers are notified if something unexpected happens without being overloaded with duplicated or unnecessary information is a key problem here. Even more difficult is figuring out how to adapt from prior events and actions so that decisions regarding management of resources can be better informed (and perhaps automated). Control theory and machine learning play a significant role in the creation of scalable monitoring and management of microservices and its associated resources.

 ©IJSRED: All Rights are Reserved

6. Failure Tolerance and Recovery

Microservices, like any other distributed system, are prone to failure. They may become unavailable, unresponsive, or malfunction for a variety of reasons, including network, hardware, or application-level issues. Any service could become temporarily unavailable to its customers due to service dependency. Communication will break occasionally in any multiuser environment. Any services could become temporarily unavailable to its customers due to service dependencies. Communication will break in any distributed system periodically. Because of the large quantity of messages travelling among the services in a microservice system, failures in communication are more probable to happen frequently. Developers must create fault-tolerant services that can smoothly respond to specific forms of breakdown or failure in order to minimise the influence of partial outages. Researchers and practitioners have developed various methods for isolating faults so that they do not spread across a distributed system. However, more work on automated failure management, as well as self-repair and self-healing technologies to repair a system after a failure, is required [11].

7. Data Management

Data management is an essential component of every programme. There are several critical factors to consider, including whether to utilise a relational database or a NoSQL database, which database provider is ideal for the application's use cases, and what sort of schema the database must have. Microservices allow you to utilise a variety of database engines. This pattern, known as database per service, has its own host of concerns. Multiple databases make maintaining them more difficult, and the company may not have a thorough understanding of the database. Previously, leveraging ACID (Atomicity, Consistency, Isolation, and Durability) transactions were simple if the monolith application utilised a typical relational database. Transactions are more difficult to manage now because there are several services, each with their very own database, and therefore more time is required to comply with the transactions. Microservices can collaborate on data consistency over time instead of transactions. This implies that modifications made by other services may not be reflected instantaneously, but they will be reflected after the information has been processed. If the user has to wait for the entire transaction to finish before exploring the data created by the dependent service, the user may not be able to do so now. The service owns the data in this scenario of a one database per service approach, and if the order service, for example, wants to understand anything regarding the invoices, it must use the invoice API. As a result, the services are loosely coupled. However, using a single database for every service is troublesome since the database structure is now closely connected. Because there is just one database, services have entry to records that must particularly be made available via requests to other services. This essentially can lead to the dropping of modularity because it's much easier to directly retrieve data from multiple databases rather than making a service call to the right service that should deliver this data.Having the order service have accessibility to other schemas such as invoices, this eventually allows order service to extract data from invoice entity without having to use the invoice service API. As a result of the close dependency during development, if the team implementing the invoicing service wishes to update the schema, they must now communicate with many other teams. The usage of a single shared database negates many of the advantages of microservices, therefore thus is not advised. Alternatively, when migrating to a microservice design, the monolith database must be divided into several databases which could only be accessible by the service that manages that business processes [13].

8. Performance

Because of the dispersed nature of microservice designs, and particularly because what would have generally been a method or function calls in a monolith are substituted with RESTful API requests over a well defined network, there are serious concerns regarding the potential performance implications. Due to the underlying fact that a couple of the microservice-based applications are now obtainable as commercial off-the-shelf products with explicitly defined performance specifications added to this worry. Developing distributed microservice-based applications has been considered as particularly dangerous since predicting their likely performance was considered as more challenging. Reporting from microservice-based systems is also a cause of concern. Since data is dispersed across multiple microservices and databases, therefore complex reporting may necessitate extensive inter-microservice communication via RESTful APIs, resulting in severe performance penalties [12].

Organisational Set of Challenges

1. Culture and Coordination of the Organisation
Having a large number of autonomous teams create and deploy services separately could be a double-edged sword. On the one hand, every team is available to make decisions without having to constantly negotiate with other teams on the issues pertaining to their own domain. On the other hand, it raises the danger of teams failing to perceive the broader

 ©IJSRED: All Rights are Reserved

picture, that is, determining if their local decisions are justified and logical in light of the system's overall design and organisation goals. This issue can take many forms, ranging from the deployment of infrastructure solutions that are hard to communicate and utilize within services to the formation of a conflict avoidance mentality wherein teams strive to solve problems that are really the responsibility of certain other teams internally. Microservice programmers will need stronger coordination structures and models as a mitigation strategy, which not only support team independence but also include the system and organisation wide objectives and goals. Some businesses (such as Netflix) have made frequent cross-team conversations a cornerstone of their corporate culture. Some companies (such as Spotify) have created applications that keep track of their services and technology in a common repository. Outside of the teams, others make decisions on technology and programming languages. Whatever approach a business chooses, these concerns must be addressed as part of the firm's culture in order to expand to hundreds or thousands of microservices [11]. If the team in charge of invoicing service has to change the schema of the invoice database, they may do so without affecting some other service as long as the API remains unchanged. It's also possible to combine all of the services into a single database. Nevertheless, using a single database for all services is troublesome since the database structure is now closely connected.

2. Unavailability of Relevant Skills

While designing separate microservices, they don't always necessitate any specific skill sets (due to the fact that they may be built using a variety of technology stacks), properly executing and administering a big application built using microservice architecture necessitates many modern expertise in distributed system development and DevOps. Since individual microservices may well be built and delivered individually, and because non-trivial systems might be made up of a bunch of them, the associated architectural sophistication (continuous development / deployment, monitoring, scaling, restoration, and so on) cannot be controlled individually. Jenkins (for continuous delivery), Docker (for container - based deployment), Docker Swarm (for container clustering and management), and Kubernetes are all notable DevOps technologies (for load balancing, scaling, and discovery). Furthermore, the basic idea that individual microservices are simple to construct owing to their relative simplicity is incorrect. Microservice programmers, in contrast to huge teams producing monoliths, where individual developers may specialise in certain technologies or features, must have a thorough awareness of a variety of non-functional elements, such as cybersecurity, system fault-tolerance, microarchitecture reliability, recoverability and latency [12].

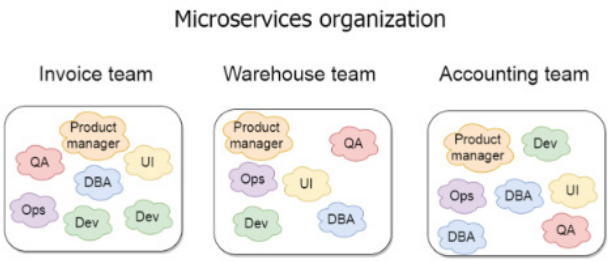
3. Governance

Microservices' dispersed nature necessitates a major departure from standard governance frameworks and practises. Microservices architecture governance is decentralised by design. Even though individual owners are accountable for their microservices, they are also responsible for evaluating the implications of alterations on other microservices that rely on them. Microservice architecture's distributed nature necessitates the use of interface / API contracts to maintain governance. Modifications to microservice interfaces have a wide range of consequences. Alterations in the user interface can be resisted and minimised, but they are essentially unavoidable. As a result, any changes to interfaces must be shared widely and integrated with associated microservices [11].

4. Structure of the Organization

The structure of the organisation is one of the organisational issues. In order to produce successful applications, the organization's structure must match the applications architecture's structure. If the business historically used monolithic applications and had large teams with defined tasks such as quality management, development, and database administration, these sets of structures will not function with microservices. According to Conway's law, the organisation that develops the system will create a system with a structure that is a carbon replica of the organization's structure. Microservices strategy does not function if the organization's structure is monolithic. These large teams must be divided into smaller teams that can function independently. In this approach, the architecture's structure corresponds to the organization's structure, and the two would not clash. The teams are experts in their fields, but when it comes to deploying business functions, they must work together. Before a delivery can be made, there must be a hand-off process. Slower development cycles are the outcome of this arrangement.

 ©IJSRED: All Rights are Reserved



When teams are structured in a microservices-based framework, they have much more deployment independence. There is no longer a third-party hand-off mechanism, and teams no longer need to wait for other teams to finish their adjustments [13].

VIII. CONCLUSION:

Fundamentally, each microservice should be bound to a specific functionality. Refactoring a monolithic system into microservices is a difficult activity as every monolith is unique and can create different hurdles during monolithic decomposition. There is no standard approach for migrating monolith to microservices. The organisation should narrow down upon the core aspects of the monolith through analysis and should select a combination of techniques which suits their needs appropriately and proceed with the migration. Irrespective of the different challenges mentioned above, if provided with the right set of organisational structure and technical flexibilities microservice architecture prove to provide great benefits in a long run depending on the overall

ACKNOWLEDGMENT

We sincerely thank RV College of Engineering and Robert Bosch Engineering and Business Solutions Private Limited for their support in the presentation of this topic. We also like to thank our friends and family for supporting us throughout the work

REFERENCES

[1] Viggiato M, Terra R, Rocha H, Valente MT, Figueiredo E. Microservices in practice: A survey study. arXiv preprint arXiv:1808.04836. 2018 Aug 14.

[2] Ghofrani J, Lübke D. Challenges of Microservices Architecture: A Survey on the State of the Practice. ZEUS. 2018 Feb 8;2018:1-8.

[3] Al-Debagy O, Martinek P. A comparative review of microservices and monolithic architectures. In2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI) 2018 Nov 21 (pp. 000149-000154). IEEE.

[4] Baškarada S, Nguyen V, Koronios A. Architecting microservices: practical opportunities and challenges. Journal of Computer Information Systems. 2018 Sep 26.

[5] Schröer C, Kruse F, Gómez JM. A Qualitative Literature Review on Microservices Identification Approaches. InSymposium and Summer School on Service-Oriented Computing 2020 Sep 13 (pp. 151-168). Springer, Cham.

[6] Villamizar M, Garcés O, Castro H, Verano M, Salamanca L, Casallas R, Gil S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In2015 10th Computing Colombian Conference (10CCC) 2015 Sep 21 (pp. 583-590).

[7]

[8]

[9]

[10]

no A, Lang M. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) 2016 May 16 (pp. 179-182). IEEE

[11] Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: The journey so far and challenges ahead. IEEE Software. 2018 May 4;35(3):24-35.

[12] Baškarada S, Nguyen V, Koronios A. Architecting microservices: practical opportunities and challenges. Journal of Computer Information Systems. 2018 Sep 26.

[13] Kalske M, Mäkitalo N, Mikkonen T. Challenges when moving from monolith to microservice architecture. InInternational Conference on Web Engineering 2017 Jun 5 (pp. 32-47). Springer, Cham.

[14] Megargel A, Shankararaman V, Walker DK. Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example. InSoftware Engineering in the Era of Cloud Computing 2020 (pp. 85-108). Springer, Cham.

[15] De Lauretis L. From Monolithic Architecture to Microservices Architecture. In2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) 2019 Oct 27 (pp. 93-96). IEEE.

[16] A. Levcovitz, R. Terra, M. T. Valente, "Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems." 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), p. 97-104, 2015

 ©IJSRED: All Rights are Reserved

[17] G. Mazlami, J. Cito and P. Leitner, "Extraction of Microservices from Monolithic Soft-ware Architectures," 2017 IEEE International Conference on Web Services (ICWS), Hon-olulu, HI, 2017, pp. 524-531. Sdf

[18] C. Fan and S. Ma, "Migrating Monolithic Mobile Application to Microservice Architec-ture: An Experiment Report," 2017 IEEE International Conference on AI & Mobile Ser¬vices (AIMS), Honolulu, HI, 2017, pp. 109-112.

[19] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595719

[20] Megargel A, Shankararaman V, Walker DK. Migrating from Monoliths to Cloud-Based Microservices: A size and complexity of the project or application being deployed.

Banking Industry Example. InSoftware Engineering in the Era of Cloud Computing 2020 (pp. 85-108). Springer, Cham.

[21] A Comparative Review of Microservices and monolithic Architectures

[22] Microservice Architectures for Scalability, Agility and Reliability in E-Commerce

[23] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including performance benchmarks into continuous integration to enable DevOps," ACM SIGSOFT Softw. Eng. Notes, vol. 40, no. 2, pp. 1–4, Mar. 2015.

[24] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," in Proceedings NetDev 0.1, 2015.

[25] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al- Hammadi, "Performance comparison between container-based and VM-based services," in 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), 2017, pp. 185–190.

 ©IJSRED: All Rights are Reserved