Monolithic to Microservices Architecture - A Framework for Design and Implementation

1st Aman Parikh
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India
aman.parikh@spit.ac.in

2nd Pranav Kumar
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India
pranav.kumar@spit.ac.in

3rd Parshav Gandhi
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India
parshav.gandhi@spit.ac.in

4th Jignesh Sisodia
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India
jsisodia@spit.ac.in

2022 International Conference on Computer, Power and Communications (ICCPC) | 979-8-3503-9784-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICCPC55978.2022.10072238

***Abstract*—Systems for scalable and reliable software applications can be realised only after a thorough consideration of the design and architecture of the underlying components. The ubiquitous use of distributed architecture to design and implement building applications serves as an impetus to delve into analysing various architectural models such as monoliths and microservices. Through the design and implementation of a modern banking system, this research paper makes a comprehensive analysis of the different aspects of monolithic and microservice architectures. The paper highlights various logistical concerns and strategies - from discussing database patterns and communication strategies to deployment methods. The paper also outlines a step-wise, algorithmic approach to aid software engineering practitioners in migrating from a monolith to a microservice. The aforementioned approach was implemented in developing the banking application, and the result was a modular, functional and scalable microservice system. The quantitative and qualitative analysis of the results were encouraging, which attests to the feasibility of the proposed methodology.**

***Index Terms*—Microservices, Distributed Computing, Software Architecture, Interprocess Communication, Monolith**

I. Introduction

The development of technology over the past 20 years and an increasingly global world has led to a staggering demand for ubiquity of Internet based services to fulfil business needs. To that end, software solutions are an absolute necessity for organisations, and the function of entire enterprises are predicated on their ability to realise their products and services on reliable, available and accessible applications. This burgeoning demand for building online software tools to solve real-world problems has in turn called for significant research and development in software systems. In order to holistically optimise for challenges relating to scale, security, ease of access and availability, researchers turned to the underlying model of different components within these systems and the complex relationships between them. This area of study was aptly termed 'Software Architecture', as it relates to a structural view of software systems, rather than a purely functional view.

The motivation to view software as almost a synthesis of its system design and component architecture can find its origins in the research work of Edsger Dijkstra, a prominent computer scientist of his time. In the thesis of his work [20], Dijkstra not only discussed the view of layering software in a sequential manner, but also emphasised the hierarchical structure of the components and the interdependence between them. This layed the foundation for profound scholarship into software architecture. Decades later, David Garlan and Mary Shaw of Carnegie Mellon University authored the book "Software Architecture: Perspectives on an Emerging Discipline" in 1996 [21]. This book was another milestone in scholarship into software systems, as it discusses and promotes concepts in software architecture such as design patterns, components, connectors and styles. Works such as [18] and [19] help us define the study of architecture in both abstract and specific terms. Different architectures are thus a product of analysing a set of features that go into building said architectures - an analysis of architectural decisions in the context of the user, system and data, principles of system design and perhaps most importantly the characteristics of architecture that the proposed model satisfies.

This research work primarily focuses on 2 architectural models - Monoliths and Microservices. Where the former is a widely-used traditional view of systems as an atomic composition of its components, the latter is a culmination of distributed systems, where different subsystems in the whole are divided and independently treated as separate processes, but synchronised with each other through some mode of communication. The traditional framework of study is to specifically analyse the models on several characteristics such as scalability, availability, integrity, security, etc. The objective of this paper is to dive deeper into the theoretical and practical aspects of both architectures and define a feasible methodology to facilitate migration from monoliths into microservices. The objective behind the same is to aid software engineers and architects to examine the architectures in the context of their system and migrate as per the requirements.

II. LITERATURE SURVEY

This section of the paper consists of selected literature mainly discussing the various aspects of distributed software systems in general, and microservices and monolithic architectures in particular. The surveyed literature established a clear picture of the current state of research into this domain and scope for further scholarship into specific aspects that can be applied to this particular research paper. di Francesco, P., et.al [2] dive into the enterprise-specific architecture of microserices, surveying literature that can answer questions regarding trends, focus of work and adoptability in the industry. A framework using the mapping method was defined in order to classify various research articles in the domain of microservice architecture. The data gathered and categorical nature of the mapping of 71 articles helped outline the relationship between research work that can further facilitate both more research work as well as industrial viability of some architecting principles.

Tyszberowicz, S., et al adopted a systematic approach to identify a suitable design for microservices based systems is defined using functional decomposition [5]. One of the key steps in building a system based on microservices, is identifying the most effective division of systems that, although are deployed individually, interact with each other cohesively. The article posits that to partition the system's state space into microservices, operational and functional relationships between system components must be identified and visualised. Furthermore, the functional decomposition algorithm was applied and tested by 3 independent teams whose findings revealed that the algorithm significantly sped up the design time.

Grzegorz et.al [6] explain through their research work that system design is a vast field, and any single principle of microservices always being superior to monoliths may not hold true. Through their work, they try to quantitatively and qualitatively compare monoliths and microservices under various different hardware and software environments, to conclude that an organisation designing a system for their software should carefully consider a suitable architecture based on their specific requirements.

In a rapid review done by Francisco Ponce et.al, they try to gather, organise, and analyse the migration techniques proposed in the literature [3]. Creating a system dependency graph and using a clustering method to design potential microservices is, in general, the most used technique. Additionally, they emphasise that while switching from a monolithic to a microservices, design is arguably the hardest undertaking.

Salanke, Girish Rao and Sanjana, G. detail the various problems arising from breaking down large systems into smaller components in the case of microservices from a communication standpoint [10]. Since communication drives functionality of services, it is important to set up a resilient pattern for the same. The proposed system is a messaging service built on distributed platforms which is highly reliable and efficient due to optimisations such as message schema validation and transformation. The service built greatly reduces code size and eases the developmental complexity of engineering the system.

Gurudat K S et.al examine the 2 broad classes of interprocess communication systems in Message Oriented Middlewares (MOMs), wherein the asynchronous message queues and synchronous REST API's are compared in their use as agents in microservice applications. The 2 patterns are adequately defined from a design perspective and implemented in a microservice system. The performance of the message queue (using RabbitMQ as broker) was found to be more optimal than that of REST API. The conclusion also provides the reason for the message queueing's better performance and feasibility, stating that no threat of packet loss and scope for using multiple queues in the channel make RabbitMQ's system a preferred choice.

III. BACKGROUND

After conducting a thorough review of the literature and the meta analysis of the domain, in these section we will be explaining the different design principles relating to software system architectures.

A. Monolithic System

In software systems, the architecture that platforms an all-encompassing, single unit of an application is referred to as "monolithic architecture". Fig 1 shows a general design of a monolithic system. Some of the characteristics of a monolith are:-

1. Tightly coupled, interdependent components within the system. The various services in the application are highly dependent on the functionality and design of other services.
2. The system does not support modularity - changes and updates made to the system reflects on the working of all the components, since they exist as a unit. Thus, independently working on or building these services is fairly challenging.
3. Monoliths allow for straightforward testing, with unit testing and integration testing simple to do over a single deployable piece of software [22]. Monitoring processes including logging and metrics calculation are very straightforward, as a server hosts just 1 unit.
4. An all-in-one software deployment also poses a risk relating to a single point of failure. Server crashes, unavailability or maintenance of any process or component within the system necessitates that the entire application may become unavailable.
5. In a system built on such architecture, a large number of services and voluminous data processing is possible only through vertical scaling i.e. increasing application size. This adversely impacts server start-time [6].

B. Microservice System

To tackle some of the negative characteristics of the Monolithic architecture, Microservices were introduced in the domain of software development. It is a collection of multiple

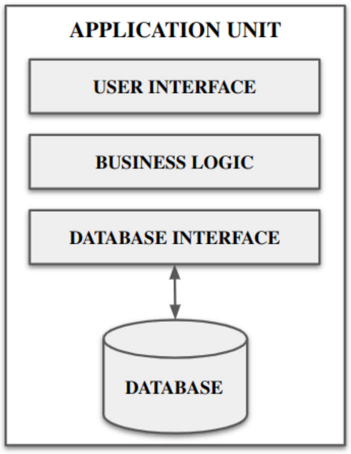

Fig. 1. A general design of a monolithic application, the user interface layer, business layer and data access and management layer, along with the database, are all packaged into a single unit.

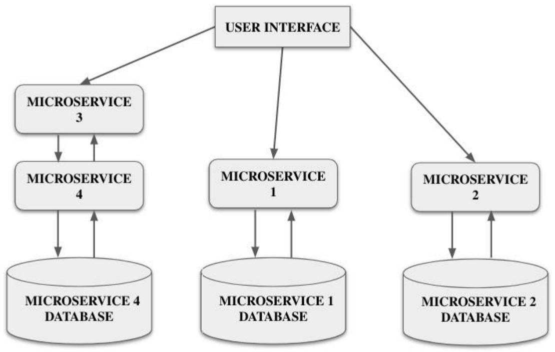

Fig. 2. A general design for a microservice system

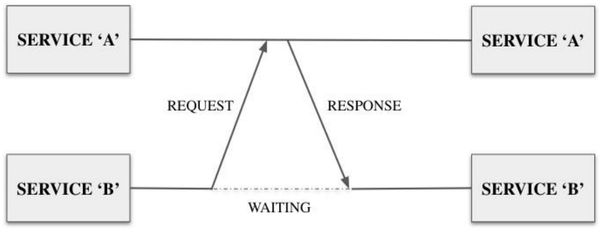

Fig. 3. How 2 services interact with one another through synchronous mode of communication.

stand-alone, independent services with a well established interservice-communication between themselves. This architectural model follows principles of distributed systems. Fig 2 shows a general shows a general design of miscroservices system. Some of the characteristics of mircroservices are:-

1. Loosely coupled, independent components in the system. The various services of the application are almost entirely independent of each other and can manage their endpoints separately

2. These services do require a medium of communicating with each other. To handle such scenarios there are multiple communication options such as RPC (Remote procedure call), Messaging queues, etc.

3. Microservices provides modular testing, each of the separate service can be tried and tested in isolation, independent of other services.

4. All the services in this architecture have their own separate codebase, database layers, CI/CD pipelines, technology stack, etc.

5. Due to the independent nature of services, there is no single point of failure. A service can be down without affecting the functionality and working of other services in the system.

6. Here scalability is not an issue. Such systems support both horizontal and vertical scaling whichever suits the requirements.

C. Interprocess Communication

In a monolithic architecture application, that usually consists of a single backend server, instances of that system are in the same process and usually communicate with each other through method calls. However, in the case of applications working as microservice systems, various services are decoupled from each other and therefore inter-service communication is one of the most crucial parts of creating microservices applications.

(i) Synchronous Communication

Synchronous communication categorises a specific type of communication in software and computing systems wherein functional data requests are followed in a strict, ordered and blocking manner. This essentially means that in a client-server model, the client program will make a request to the server program, which in turn takes some time to process the request and return a response. The client in this case will completely stop executing in the period of time that it takes to receive a response from the server. This period of time is the waiting or blocking time. One can consider such a communication pattern even in the context of microservices. Fig 3 shows the interaction of 2 services using synchronous communication.

Between microservices, in order to achieve proper ordering and access of real-time data, there are a number of options available in the case of synchronous communication. The 2 most common examples of this are (i) HTTP REST API - A stateful style of request that employs a blocking mechanism. *(ii)* gRPC - A high performance framework extension of the classic Remote Procedure Call

(ii) Asynchronous Communication

While in certain applications the requirement of syn-

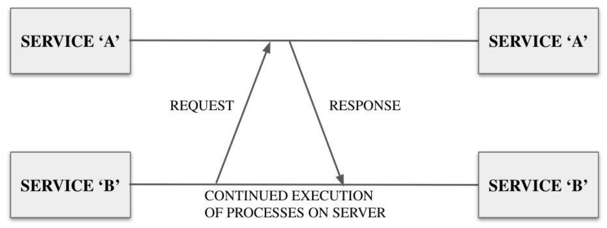

Fig. 4. How 2 services interact with one another through asynchronous mode of communication.

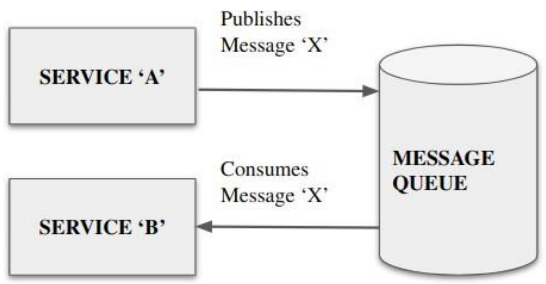

Fig. 5. Service A can push a message or task to the message queue, managed usually by some broker. The task may be received by Service B at some point, and the corresponding task may be completed.

chronous, real-time communication is unavoidable, there are many cases in which this does not hold true. In a simple client-server model, a client may make a request to the server, and this request might take some time to be processed and adequately executed. However, the client process will not stop its own execution. The running of other operations and background functions will continue in spite of response potentially not having been received. Fig 4 shows the interaction of 2 services using asynchronous commmunication.

This pattern of communication is the impetus behind message based communication between microservices where real-time transmission of data is not an absolute necessity. The message queuing system is a great example of the same, where an intermediary message broker stores 'messages' or 'tasks', that one microservice has pushed that must be consumed and followed by another microservice. Fig 5 shows the working of message queuing. These activities are not performed in real-time. The message is different from a request, and does not imply a response back to publishing service.

D. Distributed Database and the CAP Theorm

Microservice-based applications are by its design and function, distributed systems. Thus, some of the challenges pertaining to these systems fall are also considerations that engineers of a microservice architecture need to make. One of the principal challenges is in database design. Founded in 1988 by Professor Eric Brewer, the 'CAP Theorem' states that

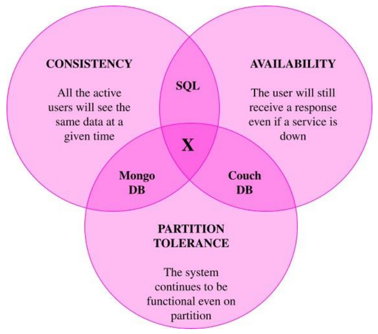
CONSISTENCY
All the active users will see the same data at a given time
SQL
AVAILABILITY
The user will still receive a response even if a service is down
X
Mongo DB
Couch DB
PARTITION TOLERANCE
The system continues to be functional even on partition

Fig. 6. The 'CAP' theorem model, as proposed by Prof. Brewer

considering a distributed system, Consistency, Availability and Partitioning cannot be all achieved at the same time as depicted in Fig 6. This theorem, along with the real-world implications are better visualised for understanding.

The loose coupling and independence of deployment and development mandate that when it comes to the management and access of data be distributed by rules of the CAP theorem.

IV. METHODOLOGY

In the work done in this research paper, a banking application was first developed on monolithic architecture. The services developed including authentication, account management, credit and debit card services and insurance registration were built on the server, written in NodeJS. The database used in the development was MySQL, a relational database that optimally supports various architecture types.

The constructed monolithic baking application, is converted to follow a microservices architecture as a part of the migration phase. All the services mentioned under monolithic have their own stand alone codebase, databases, etc. Hence, componentizing these services separately and making them independent of each other.

A. Migration

A general overview of the implemented system comprises an application facilitating various general services provided by a Bank. The entire system has been implemented in two different architectures – Monolithic architecture and Microservices architecture. Moreover, initially, the system was implemented as a Monolithic system. In the second phase of the work, the objective was to successfully convert the Monolithic system into one which is functionality wise the same as Monolithic but based on the characteristics of Microservices architecture as shown in Fig 8.

Pre-requisite - The application has been successfully created as a fully functional Monolithic system as shown in Fig 7.

(i) Identify potential microservices in the monolith

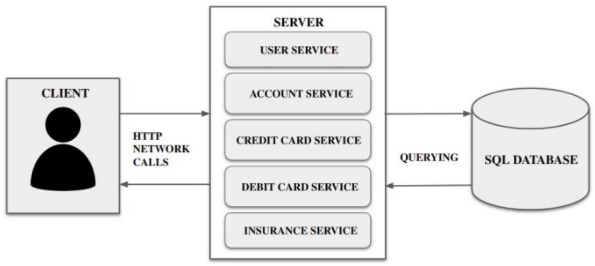

Fig. 7. Monolithic system design for our banking application

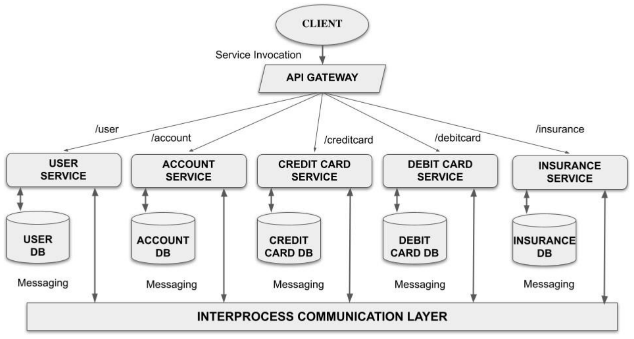

Fig. 8. Microservice system design for our banking application

In the initial step, the breakdown of the monolith by observing logical boundaries can help extract potential microservices. As per the characteristics of monolithic application, the services will inevitably be inter-dependent or inter-related by the use of relational data, computing resources and interface layer services such as middlewares. The requirement is to make note of these interdependencies amongst the services as we will have to set certain policies to handle them. For example, from our banking application point of view, among the available services, we separated out the account service and the user service and established a certain policy for them to communicate with each other while existing as separate microservices.

(ii) Design and view as separate micro-service units

In a monolithic application, the entire code base layer exists as a single unit. The routes, controllers and database interfaces (models) are all existing together remotely on the opted code repository tool like GitHub, Bitbucket, etc. In microservice architecture we will require to have separate code repositories for all the services identified from step (i). Hence, the code base layers will also be unique for all these services. For example, our monolithic banking application has a single stand alone repository on GitHub, while the microservice application has 5 different repositories for 5 different services under the same organisation on GitHub.

(iii) Database design and patterns for microservices

In the previous section, the concept of distribution of database management across the services in the context of the 'CAP' theorem was discussed. In a monolith, the entire service takes a view of the entire database, including tables relating to all services, at the same time. This approach cannot work in microservices, as it would break the founding principle of modularity and loose-coupling. While migrating to microservices, the patterns for database design and implementation has to be considered. Some of the patterns : (i) One database per service: As the name suggests, each microservice is connected in read-write access to its own microservice. Data needed across services can only be achieved through IPC, ensuring maximum modularity in the service. A drawback of this technique is that complex join queries become nearly impossible due to the loose dependencies. In processes where there needs to continuous interactions between services, too many inter-process calls become very difficult. (ii) Shared database: Another pattern is to replicate the database across each and every service. This reduces the dependence of services on IPC to read data across services, but also reduces the modularity. Logically, each microservice should be allowed read-write access only to its own corresponding data.

In application implemented in this research paper, 'One database per service pattern' was observed, as the goal was to achieve maximum modularity. The database was implemented in SQL in each service. Being a distirbuted architecture, partition tolerance was observed along with high availability (AP of CAP).

(iv) Implement strategy for interprocess communication

In the previous section regarding the major logistical challenges in building a microservice architecture, the idea of needing a reliable interprocess communication system was discussed. Asynchronous systems that provide quick, unordered IPC functionality through message queuing and task distribution are popular choices in many microservice environments, but the application discussed in this paper relies on real-time synchronicity, and guaranteed responses. However, opting for completely synchronous pattern would come at significant computational costs and potential for run-time infinite loop. Thus, in our system, a hybrid approach, consisting of asynchronous RPC was implemented. RabbitMQ's AMQP message broker provides queues in their server channels, and by designing these queues as message request and message response queues, a logical RPC method calling system could be developed. This was extremely optimised, as it used the underlying asynchronous architecture of message queues for optimality and RPC's real-time reliability for guaranteed responsiveness.

(v) Set up an API gateway

An API gateway is essentially an intermediary software application between the client and the collection of microservices. All the incoming requests to microservices from the client are diverted towards the special server, which is the only entry point for the entire microservices [8]. Unlike monoliths

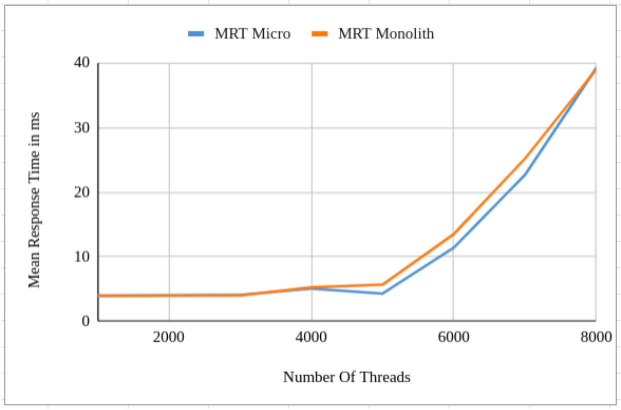

Fig. 9. Load Testing - Mean response time vs Number of threads

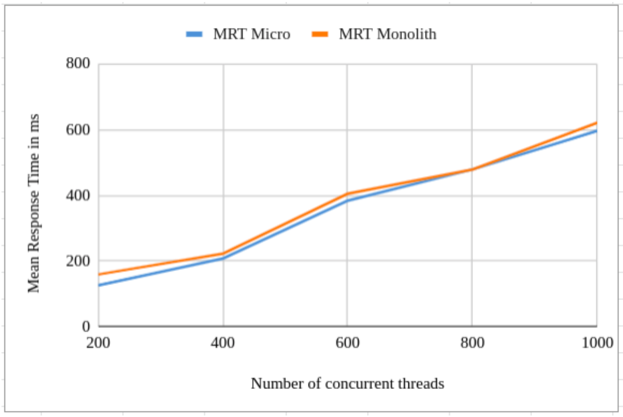

Fig. 10. Concurrency Testing - Mean response time vs Number of threads

where different functions can be called by simply altering certain parts of the endpoint URL, the same cannot be applied to microservices and a more organized routing mechanism is required. The gateway manages and channels method requests to simply access to API resources. For example, In our microservice banking application, an API gateway is implemented using AWS for facilitating API requests into resources such as the user service, account service, credit card service, etc.

(vi) Establish a deployment pattern

Deploying the application would allow external traffic onto the server. For monolithic applications, deployment can be done for the system as a whole. One of the most optimal approaches for a monolithic application is using a Serverless deployment model – cloud-native deployment model - which allows developers to expose applications to traffic without having to manually manage the servers. Pricing model for serverless hosting depends on the number of hits made to the server. In this application, the various services were deployed in a cluster through managed container images. These services were made to run in a serverless environment.

V. RESULT

The steps followed as mentioned in the previous section enabled us to successfully migrate our monolithic application to microservices architecture. The codebase was divided by structurally adding boundaries and each microservice was containerised and deployed to AWS's Elastic Container Service, with the Fargate compute engine allowing us to access the benefits of serverless compute. An API gateway was configured on AWS. The ease of use of cloud service providers allowed seamless deployment of our API's. wIn order to gauge the empirical impact of our migration, the next step was to test the applications. 2 test scenarios were enacted - load testing and concurrency testing. This was carried out on JMeter's GUI, and the docker container was run on a 16GB x64 Windows machine. The request method was a GET method in the /user/all API resource.

(a) Load Testing - 1000, 2000, ...upto 8000 user threads, with a 2-loop run and 30 second ramp-up time.

(b) Concurrency Testing - 200, 300,..upto 1000 concurrent user threads with 5 minute ramp-up and hold-time.

The result of the load test in Fig 9 indicated that while the monolith provided marginally better response time at lower thread counts, the microservice began showing quicker response times at higher thread counts near 8000. As for the concurrency test, in Fig 10, the microservice provided consistently better mean response time throughout the course of the testing phase, on average providing 1 per cent better performance.

These results might imply that microservice architecture more optimal performance. Yet this does not hold true overall in every situation, as these test scenarios cannot adequately make up for the array of qualitative differences that are considerably more crucial. The migration to microservices in our application example, allowed us to observe various database design patterns, IPC communication patterns and the working of API gateway in principle. Taking the practical nature of our proposed methodology was the key in the production of our reliable and efficient microservices.

VI. CONCLUSION

While ample research material can be found relating to the general challenges surrounding the architecture and design of microservices, this research work delves into the application specific details that confront engineering teams in practicum. A comprehensive yet modular stepwise approach was put forth, which was faithfully implemented in the development of a banking application with various dependencies and functions. The empirical and qualitative analysis revealed the microservice system was fully functional, reliable and fairly efficient. By referring to this research paper, engineering teams can examine the migration policies as well as meta-analysis presented in the background and literature survey to estimate the need to build on a microservice architecture and follow a process to enact the changes.

References

[1] O. Al-Debagy and P. Martinek, ”A Comparative Review of Microservices and Monolithic Architectures,” 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154, doi:10.1109/CINTI.2018.8928192.

[2] P. D. Francesco, I. Malavolta and P. Lago, ”Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption,” 2017 IEEE International Conference on Software Architecture (ICSA), 2017, pp. 21-30, doi: 10.1109/ICSA.2017.24.

[3] Ponce Mella, Francisco Márquez, Gastón Astudillo, Hernán, “Migrating from monolithic architecture to microservices: A Rapid Review”, 2019 38th International Conference of the Chilean Computer Science Society.

[4] Mihai Baboi, Adrian Iftene, Daniel Gˆıfu,“Dynamic Microservices to Create Scalable and Fault Tolerance Architecture ”,doi:10.1016/j.procs.2019.09.271 , .

[5] Tyszberowicz, S., Heinrich, R., Liu, B., Liu, Z. (2018). Identifying Microservices Using Functional Decomposition. In: Feng, X., Müller-Olm, M., Yang, Z. (eds) Dependable Software Engineering. Theories, Tools, and Applications. SETTA 2018. Lecture Notes in Computer Science(), vol 10998. Springer, Cham. Tyszberowicz, S., Heinrich, R., Liu, B., Liu, Z. (2018). Identifying Microservices Using Functional Decomposition. In: Feng, X., Müller-Olm, M., Yang, Z. (eds) Dependable Software Engineering. Theories, Tools, and Applications. SETTA 2018. Lecture Notes in Computer Science(), vol 10998. Springer, Cham. https://doi.org/10.1007/978-3-319-99933-3_4

[6] G. Blinowski, A. Ojdowska and A. Przybyłek, ”Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,” in IEEE Access, vol. 10, pp. 20357-20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[7] Gurudatt K S, Prof. Padmashree T, “A Better Solution Towards Microservices Communication In Web Application: A Survey”, International Journal of Innovative Research in Computer Science Technology (IJIRCST) ISSN: 2347-5552, Volume-7, Issue-3, May-2019 DOI: 10.21276/ijircst.2019.7.3.7.

[8] Zhao, J Jing, S Jiang, L. (2018). Management of API Gateway Based on Micro-service Architecture. Journal of Physics: Conference Series. 1087. 032032. 10.1088/1742-6596/1087/3/032032.

[9] Shafabakhsh, Benyamin Robert, Lagerström Hacks, Simon. (2020). Evaluating the Impact of Inter Process Communication in Microservice Architectures, 8th International Workshop on Quantitative Approaches to Software Quality.

[10] G B, Sanjana Salanke N S, Girish Rao. (2021). High Resilient Messaging Service for Microservice Architecture. International Journal of Applied Engineering Research. 16. 357. 10.37622/IJAER/16.5.2021.357-361.

[11] S. Kul and A. Sayar, ”A Survey of Publish/Subscribe Middleware Systems for Microservice Communication,” 2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), 2021, pp. 781-785, doi: 10.1109/ISMSIT52890.2021.9604746.

[12] Nishanil. “Communication in a Microservice Architecture.” Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[13] C. Kong et al., ”ACID Encountering the CAP Theorem: Two Bank Case Studies,” 2015 12th Web Information System and Application Conference (WISA), 2015, pp. 235-240, doi: 10.1109/WISA.2015.63.

[14] S. Gilbert and N. Lynch, ”Perspectives on the CAP Theorem,” in Computer, vol. 45, no. 2, pp. 30-36, Feb. 2012, doi: 10.1109/MC.2011.389.

[15] S. Zepeda and A. Nuñez, “Asynchronous communication for Improved Data Transport in the network,” Journal of Physics: Conference Series, vol. 1828, no. 1, p. 012078, 2021.

[16] Bass, L., Clements, P. and Kazman, R., 2003. Software architecture in practice. Addison-Wesley Professional.

[17] Hofmeister, C., Nord, R. and Soni, D., 2000. Applied software architecture. Addison-Wesley Professional.

[18] M. Villamizar et al., ”Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” 2015 10th Computing Colombian Conference (10CCC), 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

[19] Edsger W. Dijkstra, “The Structure of the ”THE”-Multiprogramming System”, 1967. ACM Symposium on Operating System Principles.

[20] Mary, S. and David, G., 1996. Software architecture: Perspectives on an emerging discipline. Prentice-Hall.

[21] “RPC with Rabbitmq direct reply-to,” CloudAMQP. [Online]. Available: https://www.cloudamqp.com/blog/rpc-with-rabbitmq-direct-reply-to.html.

[22] Dey, S., 2020. Microservices Architecture and its Testing Challenges. TestProject. Available at: https://blog.testproject.io/2020/12/07/microservices-architecture-and-its-testing-challenges/.