International Journal of Emerging Trends & Technology in Computer Science (IJETTCS)
Web Site: www.ijettcs.org Email: editor@ijettcs.org, editorijettcs@gmail.com
Volume 11, Issue 4, July – August 2022 ISSN 2278-6856

A STUDY ON BUILDING MICROSERVICES

G Aravind[1] and M Nishanth Gopal[2]

[1 2] R V College of Engineering, Department of Electronics and Communication Engineering, Bengaluru 560059, India

Abstract: ***A new paradigm that is paving the way for many methods and standards is the cloud. Due to the demands of the cloud, architectural styles are also changing. Microservices are now regarded as the preferred design for scalable, quickly changing cloud systems. A thorough mapping study of microservices was conducted for this research. It aims to identify current microservices trends, the driving force behind microservices research, upcoming standards, and potential research needs. Researchers and practitioners in the field of software engineering who want to be informed of emerging trends regarding SOA and cloud computing can benefit from the results that have been collected.***

1. INTRODUCTION

The concept of service-oriented architecture (SOA) has become popular for creating distributed systems with independent services as the constituent parts. Services are fundamental components that are independently developed and made available online. To communicate between various computers for services, standard internet protocols are employed. Since SOA offers many benefits for simple and affordable distributed software system development, it is the most popular solution for interoperability in the modern internet environment. In order to create reliable and reusable services while taking into account the requirements and peculiarities of this technology, service-oriented software engineering refers to the advancement of current software engineering methodologies. Service-oriented computing (SOC) is a paradigm that uses services as the core components of applications development. So, the goal of service-oriented software engineering is to use software engineering approaches to design and construct service-based applications that adhere to the SOC paradigm and SOA [1] principles. The development, deployment, and upkeep of the microservices are done individually. This gives the teams the freedom to choose the technology that best suits the demands of the present business behaviour. Depending on the microservice, the language and database may vary. Instead of exchanging data, they communicate with one another via the Representational State Transfer (REST) protocol. The flexibility, autonomy, scalability, robustness, and simplicity of continuous deployment are the most significant advantages of employing microservices. Microservices were first mentioned at in 2010, although the definition of a microservice used there does not entirely correspond to the definition of a microservice used today in literature. According to a 2010 study, microservices are small, REST-based systems.

2. MOTIVATION

As applications are developed as aggregates of little services, each executing in its own process and communicating with one another through simple mechanisms, the microservice building is quickly becoming the industrial standard. Netflix is one of the top companies using the microservice architectural technique, processing over 500 microservices and two billion API edge responses daily. However, Uber has around 1300 microservices to increase scalability and robustness.
In the area of microservices, there are several intriguing reviews. In order to analyse the growing standards for microservices, also look at the forms of study done and the practical reasons for deploying the microservices design. There are more industry-based suggestions for microservices architectural patterns, such as patterns languages and others; however, for the sake of this review, we have emphasized on academic articles. The earlier publications presented the available information regarding microservices, and yet none of them discussed the architectural strategies and patterns suggested for this field. This provides us with the incentive to carry out this systematic review.

3. MICROSERVICES VS. MONOLOTHIC ARCHITECTURE

A software program constructed as a cohesive entity that is independent of other applications and self-contained is referred to as having a monolithic architecture. A monolith architectural for software engineering isn't far from what the name "monolith" is frequently associated with: something enormous and glacial. A monolithic design unites all of the commercial interests in a single, sizable computing network with a single code base. This kind of application requires changing the complete stack, which requires accessing the code base, creating an updated service-side interface, and deploying it. Updates becomes complicated and time-consuming as a result.
A collection of independently scalable services are the foundation of a microservices architecture, commonly referred to as microservices. These services have a distinct objective or their own application logic and database. Each service undergoes updating, development, deployment, and scalability. Major business and domain-specific [2] concerns are decoupled via microservices and placed in different, independent code bases. By breaking down work

into smaller activities that run independently of one another and add to the achievement whole, microservices make whatever complexity visible and easier to manage, without reducing it.

Many projects begin as monoliths prior transitioning to microservice architectures. It may tend to become problematic to have numerous developers regarded as a single codebase as newer devices are added to the a monolith. Code conflicts rise in frequency, and there is a greater chance that enhancements to one feature will disrupt another. If these unpleasant patterns start to show, it could be time to migrate to microservices.

4 PRINCIPLES OF MODELING MICROSERVICES

Over the past ten years, distributed systems have evolved from large, code-intensive monolithic apps to more compact, independent microservices [3]. However, creating these systems has its own set of challenges. This paper presents a comprehensive approach to the issues that appropriate tools and operators must take into account when creating, managing, and upgrading microservice architectures, with lots of examples and helpful guidance.

Technologies for microservices are developing swiftly. While delving into the most recent approaches for modelling, integrating, evaluating, deploying, and controlling your own autonomous services, [4] gives a solid foundation in the principles. Throughout the paper, one will follow a hypothetical corporation to discover how creating a microservice architecture impacts a single domain.

This work will thorough three key ideas that one should keep in mind when modelling microservices in this essay. Modeling microservices is the first step in creating them. This aids in determining the microservices' scope. At this initial stage, a lot is on the line. Inaccurate modelling can result in catastrophic failures when developing software. The three magic principles this work focuses on are single responsibility, high cohesion and Loose coupling.

4.1 High Cohesion

High cohesion is a term used in software engineering to describe how closely all the classes' or routines' code supports a main objective. High cohesion classes are those that have features that are closely related to one another; the logical goal is to maximise cohesiveness.

Cohesion, then, is the extent to which a section of a codebase functions as a logically cohesive and atomic unit. Therefore, maintaining linked components of a codebase in a single location is the essence of high cohesiveness. Make very sure that every software has strong cohesiveness as a general rule.In accordance with the high cohesion idea, "all linked conduct should sit together".

Consider that we offer the services "service management" and "total service calculator" While one is in charge of determining total amount of services and other one is responsible for service management. Let's say the "get service details" API returns the service data, which includes the information service fees. The total is determined by the calculator API using the per unit service pricing. There is a chance that the arithmetic service will also be updated whenever there is an attribute change in the service[5]. Avoiding this tight connection is necessary. The computation logic should integrate with the service management since it only relates to services required. We can attain high cohesiveness in this method since the relevant behaviour will be grouped together.

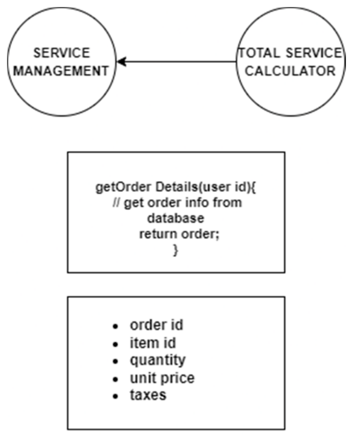

Figure 1 : Sample of High Cohesion

4.2 Loose Coupling

Loosely connected system is one where the each of its constituent parts has little to no understanding of, or uses, the definitions of, other independent constituent parts. The coupling between classes, interfaces, objects, and services is one of the subregions. In a nutshell, loose coupling in a microservice architecture refers to the idea that individual services should be independent of one another and should not be aware of one other's existence.In order to prevent the alteration in one service from affecting other services, loose coupling is used. This relies on a variety of variables. Tight coupling [6] occurs throughout the services when single competence and maximum cohesion standards are not followed. One of the key tenets of a successful microservice architecture is loose coupling. Although free coupling has all these advantages, it might be difficult to achieve in practise. Of course, not all connection in the system can be removed; some coupling is completely acceptable as long as it doesn't compromise the intended result.

4.3 Single Responsibility Principle

Each microservice should fulfil a single obligation because the principle is self-explanatory. When developing microservices, this idea is among the most crucial design principles. For optimal agility, it aids in defining the service's scope and boundaries. Abandoning this idea could result in the creation of another monolithic system. According to the single responsibility concept, a class should only have one reason to change. Although this might initially seem terrible, it actually serves as motivation to "split" the classes into smaller segments according to their functional roles inside our system. By doing this, we may isolate a certain "region of change" in our software based on the requirements of the class that is being modified. Furthermore, if we properly "dissect" our code, we will typically find that each little class has fewer dependencies than the preceding "bigger" class.

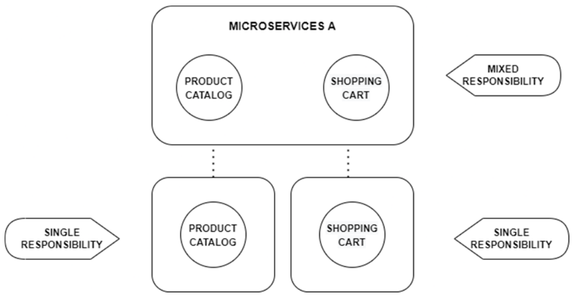

Figure 2 : Sample of Single Responsiblity Principle

5 MIICROSERVICE ARCHITECTURE

Loose coupling is the key element between services offered in a micro services architecture. Each service may have its own database and servers, or it may share one, relying on its technological needs. Because of this, the development of reusable components may vary depending on the application's scalability[7] needs. Along with appropriate communication protocols, a reliable microkernel architecture should also include economical deployment techniques.

The following are some of the most widespread architectural patterns, deployment methodologies, communication, and monitoring mechanisms.

5.1 Mechanisms for Communication

Micro services need a proper yet sophisticated route for interservice or synchronisation because they have been made up of a number of small services . Two main types of asynchronous communication systems be derived from this.

- Asynchronous : In this case, the customer does not in itself watch for a service response. The service might serve, it would do so asynchronously.
- Synchronous : This kind of approved person on the request and responder model. After submitting a request, the client waits, possibly even blocking other clients, until it receives a response. For communication between the process to run well, a quick and timely response is necessary. This can be achieved through the use of the REST (Representational State Transfer) or HTTP (Hypertext Transfer Protocol) protocols.

Clients can interface with the application that used an API (Application Programming Interface) Gateway for communicating. As an interface, this API Gateway sends customer questions to the area network.

5.2 Observation

Because micro companies are decentralized in origin, it is crucial to monitor both their resource usage and performance. Monitoring the system's availability and taking preventive action are extremely crucial to avoiding more failures. Microservices are monitored in more ways than just how well they work. Watching also takes into account the system's well being, user tracking and identity for security monitoring, oversight of aberrant system occurrences and behaviours, and adequate availability for keeping SLAs (Service Level Agreements).
The performance appraisal of these services presents a significant difficulty while individual services are separate of one another and occasionally employ different technologies. Additionally, runtime service monitoring is required for better defect detection. To prevent erroneous anomaly detection, the system's behaviour change also needs to be checked carefully. Tracking of data is consequently essential for designing performance models for various environments, as it can aid in project identification and the creation of some efficient deployment methods.

5.3 Observation

The identification of a complex integrated is an important factor of developing the micro services architecture. The proper creation and operation of the micro services require on the coordination of their various components. Depending on the details of the target purpose, micro services can be created utilising a variety of strategies in contrast to the patterns[8]. In order to aid developers in finding an appropriate substitute, several architectural models can also be categorized together related to the product traits like orchestration, migration, communication, design, and so on.

5.4 Deployment techniques

In addition to offering and growing micro services, management and maintenance are key attributes. So that each micro service's resources and technological stack fluctuate, so do its requirements. This creates a variety of possible solutions for delivering micro services.

- Single Application per Host Deployment Pattern: In this architecture, the service instances would not share hosts or resources. On a given host, each service instance is installed. This host could be a physical machine or even a container. It is possible to operate the service instances using container technologies like Docker. Better resource monitoring is made available by the separation of service instances. However, under this architecture, resource use, performance, and scalability are all hampered.

- Multiple Services per Host: Micro services were provided on a variety of physical or virtual hosts. The same infrastructure and operating system will be used by different service instances in this approach. One of the key advantages of this style is the efficient use of resources. However, by utilising this paradigm, performance and scalability can be enhanced. Monitoring the utilisation of individual service instances may also create some complications.

The serverless deployment strategy is another approach for deploying web applications that some applications can makes use of.

6 MIICROSERVICE ARCHITECTURE

The major challenge facing services and preserving data integrity results in the increase of services expands, making microservices architecture one of the best architectural patterns but one that presents some of the most significant difficulties. Due to reliance on other services, monitoring also becomes a bottleneck.

The distributed nature of this data analysis strategy presents some difficulties. First, there can be duplicate data across the data storage, meaning that the identical piece of information might appear more than once. For monitoring, reporting, or archiving purposes, data [9] may be kept as part of the implementation process and later transferred to another location. Data consistency and integrity problems may arise due of duplicate or fragmented data. Traditional data management methods can also be used to enforce data relationships that span different services. The developers must make sure none of the possibilities the microservice can collapse will bring the system to a halt in order to achieve availability, which is the ultimate goal. Therefore, developers must be aware of all possible failure mechanisms and prepare backups. She stated that robust resilience testing, which includes chaos testing, stress testing, and code testing among other proactive tests, is essential for effective managing disasters. Every failure mode needs to be put into use to test how it stands the test of time.

Communication is necessary for any adjustments as the quantity of information grows so that reliant services can improve. This needs accurate API documentation. An agreement between a service and its customers is expressed in a microservice API. The contract is crucial since you can only evolve a functionality independently if you don't violate its API agreement. Your API Gateway or client applications will be damaged if you modify the contract.

The organisation makes a huge system with a thousand distinct ways to do anything when architects and developers create a microservices structure [10] employing different languages, individual infrastructures, and launching custom scripts. It might end up having hundreds of services, that are active, many of which are kept up to date, and others of which are neglected. The new development team and core elements are overworked by this.

A microservices-based program has several separate services, in contrast to a large system, whose implementation and management appear to be simpler because of central control and monitoring. The operating team's tiresome work is overseeing multiple services. This suggests that considerable work is required to guarantee the built application's resilience and prevent any failovers. Especially compared to a single process that may ordinarily do equivalent work while having a high throughput, such an aspect doubles its operational overhead.

7 CONCLUSION

A distributed design strategy called microservices architecture aims to get beyond the drawbacks of conventional monolithic structures. Microservices speed up cycle times while enabling the scalability of enterprises and applications. They do, however, also have a few drawbacks that could increase takes a lot of effort and operational load. By using a divide and conquer methodology in the design and deployment of software, microservice architecture offers a variety of advantages. Microservices can be upgraded and replaced, or they can be isolated, expanded up or down. The requirements and advantages of the deployment might both be impacted by the modularity of microservices. The client setting and application needs are wholly responsible for determining the optimum solution, which is not universal. This reference design is focused on enterprise microservices that aren't even immediately disclosed with the outside world after offering a full overview on microservices and also some of the elements that determine a client's demands and cost to profit metrics.

References

[1] C. Pahl and P. Jamshidi, “Microservices: A systematic mapping study,” 01 2016, pp. 137–146.R. Caves, Multinational Enterprise and Economic Analysis, Cambridge University Press, Cambridge, 1982.

[2] A. Gallidabino, C. Pautasso, V. Ilvonen, T. Mikkonen, K. Systa, J.- ¨ P. Voutilainen, and A. Taivalsaari, “On the architecture of liquid soft ware: technology alternatives and design space,” in 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE, 2016, pp. 122–127.

[3] P. Goodman and A. Groce, “Deepstate: Symbolic unit testing for c and c++,” in NDSS Workshop on Binary Analysis Research, 2018.

[4] M. H. Valipour, B. Amirzafari, K. N. Maleki, and N. Daneshpour, “A brief survey of software architecture concepts and service oriented architecture,” in 2009 2nd IEEE International Conference on Computer

Science and Information Technology, 2009, pp. 34–38.

[5] R. Amit and N. G. Cholli, “Application of lean principles in software development processes,” in 2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENT CON), vol. 1. IEEE, 2021, pp. 38–42

[6] N. Alshuqayran, N. Ali, and R. Evans, “A systematic mapping study in microservice architecture,” in 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 2016, pp. 44–51.

[7] V. Chayapathy, G. Anitha, and B. Sharath, “Iot based home automation by using personal assistant,” in 2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon). IEEE, 2017, pp. 385–389.

[8] T. Cerny, M. J. Donahoo, and M. Trnka, “Contextual understanding of microservice architecture: current and future directions,” ACM SIGAPP Applied Computing Review, vol. 17, no. 4, pp. 29–45, 2018.

[9] T. Yarygina and A. H. Bagge, “Overcoming security challenges in mi croservice architectures,” in 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2018, pp. 11–20.

[10] “Deployment and communication patterns in microservice architectures: A systematic literature review,” Journal of Systems and Software, vol. 180, p. 111014, 2021.