Transitioning from Monolithic to Microservices Architectures: Evolution, Comparison, and Case Study on Netflix

Mr. Shantanu Patil " , Prof Flavia Gonsalves**
Institute of Computer Science, Mumbai Educational Trust-MET ICS, Mumbai 400068, India
mca22_1335ics@met.edu

***Abstract*— The transition of software architecture from monolithic to microservices represents a significant shift and evolution in the life cycle of development and deployment. We hereby present the basic differences between monolithic architectures and microservices, discussing the advantages and challenges associated with each, and give the detailed case study of Netflix, as a leading company in the implementation of microservices. In this respect, it is interesting to look at how the implementation went on, what wins were observed, and what lessons were taken. The paper seeks to provide an all-round understanding of how microservices can fix the places where monolithic architectures have failed, therefore offering insights for an organization contemplating a similar transition.**

***Keywords*— monolithic, microservices, Netflix**

I. INTRODUCTION

In the rapidly evolving domain of software development, architectural patterns constitute a foundation for the development of scalable, maintainable, and efficient applications. The traditional form of architecting software has been monolithic, where a single unified codebase contains all the functions of the application. However, as applications grow in scale and complexity, the monolithic approach begins to show its weaknesses-such as scaling, maintaining, or deploying the software.

The emergence of microservices architecture is creating a paradigm shift in the way the design, management, and implementation of modern software systems are conceptualized. Microservices distribute an application into a set of small and loosely coupled services which may be designed, deployed, and scaled independently of each other. This new paradigm promises flexibility, resilience, and scalability and is therefore an ideal choice in modern software development.

This paper presents the change from monolithic to microservices architecture, which is done by analyzing and comparing both of their main characteristics, and also looking at Motives behind such a change. We also present a detailed case study on Netflix, one of the first and largest adopters of microservices, allowing a more practical experience exposing benefits and lessons learnt that an organization should take into consideration during deciding factors for such a change.

A. History

1) Monolithic Architecture: Monolithic architecture has been the traditional model used for designing systems of software. All components and functionalities of an application were closely glued together into a single codebase. This also consisted of the user interface, business logic, and even data access layers. Although monolithic architecture makes things easier to develop and deploy during the early stages, it usually poses some major challenges as the application grows:

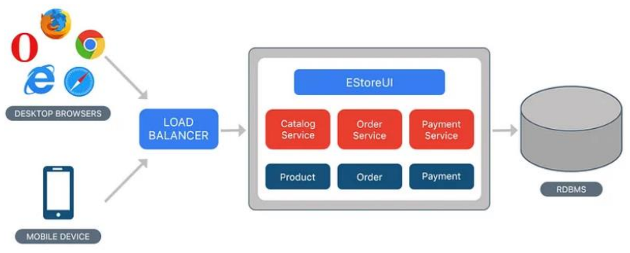

Fig. 1. Monolithic Architecture

a) Scalability: A monolith application normally scales very complex and resource-dependent. Since components are dependent on one another, scaling typically involves multiple deployments of this full application, which is inefficient.

b) Maintenance: A very large monolithic codebase is pretty hard to manage. Small changes within one part of your application may require a lot of testing and redeployment of the full system, heightening the risk of bugs being introduced.

c) Flexibility: Monolithic applications are not flexible in the event of changes. If one component needs to be updated or replaced, the whole application needs to be changed, making the development cycle slow and less inclined towards innovation.

*d) Deployment:*Continuous deployment and integration are not easy in a monolithic framework. When a change is made, whether small or large, the whole application needs to redeploy, which means bringing down the application and affecting the services.

2) Evolution to Microservices Architecture: As much as the limitations brought about by monolithic architecture in the development of software have necessitated an admiration of Microservices Architecture as an alternate, appreciably modular approach to building systems of software. By definition, Microservices Architecture is the practice that completes breaking an application into a series of small independent services focused on specific and precise functionality. The services communicate with one another only through well-defined APIs*:*

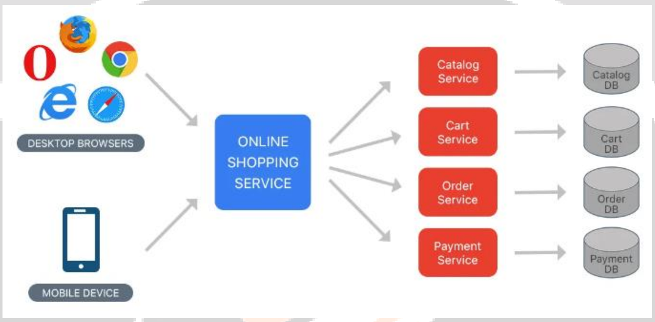

Fig. 2. Microservices Architecture

a) Scalability: The ability to scale microservices in a way depending on demand is one of the most obvious benefits. Such fine-grained scaling fits perfectly with resource optimization and ultimately better performance.

b) Maintenance: Every microservice can be developed, tested, and deployed independently. Separation of such concerns brings a decreased risk of far-reaching issues and simplifies maintenance.

c) Flexibility: This architecture delivers room for innovation. Microservices allow teams to try new technologies or methodologies within individual services without affecting the whole system.

*d) Deployment:*Continuous integration and continuous deployment pipelines are easier with microservices. Because services are independent, they can be deployed or updated without downtime-better, faster, and more reliable releases.

II. Comparative Analysis

1) Scalability:

Attribute	Monolithic Architecture	Microservices Architecture
Scaling Approach	Vertical Scaling (adding more resources to a single server)	Horizontal Scaling (adding more instances of services)
Resource Utilization	Often inefficient, as scaling requires duplicating the entire application	More efficient, as only the necessary services are scaled

Attribute	Monolithic Architecture	Microservices Architecture
Example	Scaling a monolithic e-commerce application means increasing the resources (CPU, memory) for the whole application, even if only the product search functionality needs more capacity	Amazon: Scales individual microservices independently, such as separate services for product search, payment processing, and user management

2) Deployment :

Attribute	Monolithic Architecture	Microservices Architecture
Deployment Unit	Single deployment unit	Multiple independent deployment units
Frequency	Slower, as the entire application needs to be redeployed	Faster, as individual services can be deployed independently
Risk	Higher, because changes in one part can affect the whole application	Lower, as changes are isolated to individual services
Example	Traditional banking systems: Any update or feature addition requires the whole system to be redeployed, increasing the risk of downtime	Netflix: Continuously deploys updates to individual microservices without affecting the entire system, allowing for rapid feature releases and bug fixes

3) Fault Isolation:

Attribute	Monolithic Architecture	Microservices Architecture
Failure Impact	Failure in one component can potentially take down the entire application	Failures are isolated to the affected microservices, minimizing the impact on the overall system
Recovery	More challenging, as diagnosing and fixing the problem involves the whole application	Easier, as faults can be isolated, diagnosed, and fixed within individual services
Example	A failure in the payment processing module of a monolithic e-commerce application could bring down the entire application	Spotify: Uses microservices to ensure that if the recommendation service fails, it doesn't impact the music streaming service, providing a better user experience despite partial failures

4) Performance :

Attribute	Monolithic Architecture	Microservices Architecture
Internal Communication	Faster, as all components communicate within the same process	Potentially slower, due to network latency and inter-process communication
Optimization	Optimizing one part of the application can be difficult without affecting others	Individual services can be optimized independently

Attribute	Monolithic Architecture	Microservices Architecture
Example	In a monolithic CRM application, optimizing database access might require changes across the whole application, risking regressions	Uber: Optimizes its ride-matching service separately from other services like payments or notifications, allowing for targeted performance enhancements without widespread risk

5) Development and Team Organization:

Attribute	Monolithic Architecture	Microservices Architecture
Team Structure	Typically organized around technical layers (e.g., frontend, backend, database)	Organized around business capabilities (e.g., product search, user management)
Development Speed	Slower, as changes in one part require coordination across the whole team	Faster, as teams can develop and deploy their services independently
Example	Traditional ERP systems often require coordinated efforts from multiple teams for any significant change, slowing down development	Amazon: Uses “two-pizza teams” to develop individual microservices, allowing for rapid development and deployment cycles

III. NETFLIX

1) Background:

Netflix was founded in 1997 and it started as a service for renting DVDs but soon evolved into one of the largest streaming services in the world. The growth of Netflix had to overcome several technical hurdles chiefly due to its monolithic architecture. In 2008, Netflix started having major service outages. It couldn't scale its monolithic architecture that was unable to handle the growing traffic. Then came the decision to explore other architectures, which led to its own microservices.

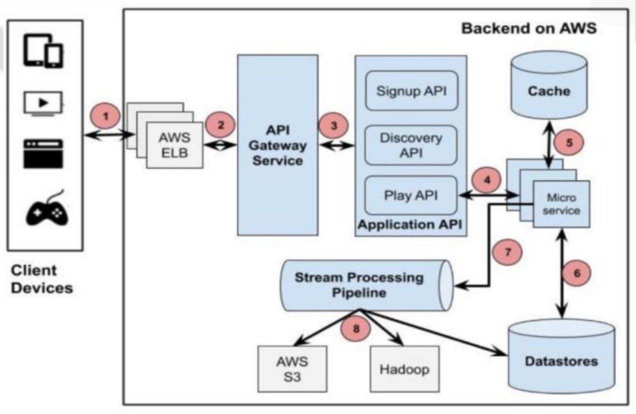

Fig. 3. Netflix’s Microservices Architecture

2) Motivations for Transitions: There were several major reasons that made the Netflix move from monolithic to microservices architecture as follows:

a) Scalability: Since there was a continuous rise in the number of users and the data, the monolithic architecture could not deal with that effectively. Scaling simply required making a copy of the entire application, which wasted resources and increased the cost of operations.

b) Resilience: Service outages effected the entire applicatiion when the system was monolithic. Netflix "wanted an architecture where failures would be isolated so the entire service wouldn't come down because of it.

c) Development Velocity: The monolithic codebase resulted in slow development cycles. Development, testing, and deployment of features independently were tricky. Therefore the rate at which Netflix could introduce innovations and new features wasnt as fast as they could.

d) Global Expansion: As it expanded globally, Netflix required an extremely flexible and scalable architecture that would meet different network conditions in different geographical locations and regional demands.

3) Transition Strategy: Netflix transitioned to microservices from its monolithic architecture in a step-by-step fashion. The following were the major characteristics of such an approach:

a) Service Decompositions: Netflix initially determined and isolated the major functional areas of its monolithic application which could be developed as independent microservices. A few of the initial services included the services for managing users, the movie catalog, and the recommendation engine.

b) API Gateway: The first one was the introduction of an API Gateway to assist in handling communication between the client and microservices. The gateway performed the function of routing, composition, and also protocol translation thus enabling the clients to deal with more than one service in a very smooth manner.

c) Decentralized Data Manageement: This is a decentralized manner of managing data. Each of the microservices manages its database resulting in consistent data and independent services.

d) Automated Deployement and CI/CD: Netflix spent millions on automation in order to cope with microservices deployment. Continuous Integration and Continuous Deployment pipelines were created. This facilitated speedy and reliable delivery of changes in code.

e) Monitoring and Logging: With a microservices architecture the level of ;complexity goes manifold. Thus, Netflix developed an elaborate monitoring and logging apparatus. This provided real insight into real-time performance. Issues could be brought to light quickly and fixed.

4) Challenges and Solutions: While transitioning, Netflix had to face various issues and developed pathbreaking solutions for the same:

a) Inter-Service Communication: When handling such a vast array of microservices, the internal communication among them introduced its own level of complexity. Netflix had to implement and leverage light weight protocols like REST and later gRPC for effective communication between services. Hystrix, a latency and fault-tolerance library, helped in developing fault-tolerant services by Netflix.

b) Data Consistency: Data consistency was a major challenge confronting distributed services. For solving the problems of data replication and consistency, Netflix had to use eventual consistency models with distributed data stores like Cassandra.

c) Service Discovery: Having a large number of services up and running, it was equally important to find the right service instances and connect to the right instances. In an effort for service registration and discovering to take place dynamically in a runtime environment, Netflix developed what is called today as Eureka.

d) Deployment and Versioning: The deployment and versioning of microservices called for keen planning. Docker containers, along with Kubernetes for container orchestration, provided an easy way to deploy and scale services.

e) Security: the inter-service communication and data became very important. Netflix had in place very strict security measures, from mutual TLS for service to service communication, and coupling this with tight access controls.

5) Outcomes: The move to microservices brought several significant benefits for Netflix:

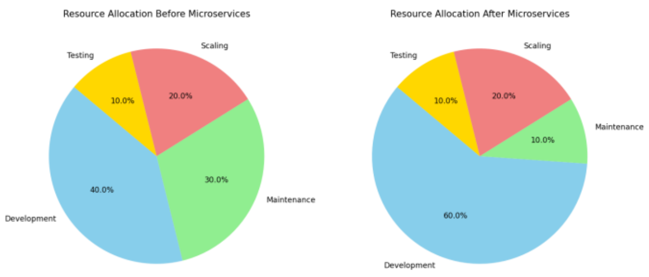

Fig. 4. This pie chart compares the resource allocation (e.g., development, maintenance, scaling) before and after Netflix's transition to microservices

a) Improved Scalability: Microservices allowed Netflix to scale components individually, thus achieving increased efficiencies in the utilization of resources and improved performance under high loads.

b) Enhanced Resilience: The microservice architecture increased resiliency of the systems at Netflix. The failure of a service did not affect the entire application. This meant that Netflix had increased availability and reliability for its services.

c) Faster Development Cycles: Independent development, testing, and deployment of microservices brought much faster development cycles to Netflix, and the company could deliver new features and improvements much more quickly.

d) Global Reach: Microservices flexibility and scalability supported the global reach for Netflix. It was possible to adapt Netflix services to regional requirements and network conditions in all different places around the world.

e) Innovation and experimentation: Microservices allowed Netflix to try and experiment with brand new technologies and methodologies for individual services without jeopardizing the entire system's stability.

6) Lessons Learned: Netflix's journey from a monolith to microservices offers numerous learning points that the organization can pursue:

a) Incremental Transition: Divide the journey from monolithic to microservices into finite steps to decrease the risk and more step-by-step learning curve

b) Invest in Automation: Invest in automation deployment, testing, and monitoring since microservices add more complexity

c) Robust Monitoring: Monitor and log all your services extensively. Without decent monitoring, there's difficulty in identifying real-time service performance problems.

d) Designed for Failure: Both designing and building resilience into the architecture from the very beginning helps graceful handling of failures.

e) Decentralize Decision Making: Decentralize decision making empowers the teams that are managing different services to take their decisions.

IV. Conclusion

The transition from monolithic to microservices architecture represents a significant evolution in software design and development. This shift addresses many of the limitations inherent in monolithic systems, such as scalability, maintainability, and deployment challenges. Through the decomposition of applications into smaller, independent services, microservices architecture offers enhanced scalability, resilience, and flexibility, enabling organizations to innovate and respond to changing market demands more effectively.

The detailed case study of Netflix illustrates the practical benefits and challenges associated with transitioning to microservices. Netflix's experience highlights several critical factors for successful implementation, including the importance of incremental adoption, strong DevOps culture, robust monitoring and logging, and effective API management. These insights provide valuable guidance for other organizations considering a similar transition.

The literature review further underscores the advantages and complexities of microservices architecture, drawing on comparisons with monolithic systems and examining case studies from industry leaders like Amazon, Uber, and Spotify. The findings emphasize that while microservices offer significant improvements in development agility and system resilience, they also require sophisticated management practices to handle the increased complexity of distributed systems.

As the software industry continues to evolve, the adoption of microservices architecture is likely to become more prevalent. Organizations looking to leverage the benefits of microservices must carefully plan their transition, adopt best practices, and invest in the necessary tools and cultural shifts to manage the complexities of this architectural approach. The lessons learned from pioneers like Netflix provide a valuable roadmap for navigating this transformation and achieving long-term success.

REFERENCES

[1] Brikman, Y. (2016). *Microservices at Spotify: Lessons Learned*. Retrieved from https://www.ybrikman.com/writing/2016/04/18/microservices-at-spotify-lessons-learned/

[2] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering* (pp. 195-216). Springer.

[3] Farquhar, A. (2015). *Microservices: Decomposing Applications for Deployability and Scalability*. Retrieved from https://www.infoq.com/articles/microservices-intro

[4] Fowler, M., & Lewis, J. (2014). *Microservices: A definition of this new architectural term*. Retrieved from https://martinfowler.com/articles/microservices.html

[5] Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. IT Revolution Press.

[6] Lewis, J., & Fowler, M. (2014). *Microservices*. Retrieved from https://martinfowler.com/articles/microservices.html

[7] Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc.

[8] Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc.

[9] Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, 4(5), 22-32.

[10] Thönes, J. (2015). Microservices. *IEEE Software*, 32(1), 116-116.

[11] Varia, J. (2007). *Amazon Web Services: Architecting for the Cloud*. Retrieved from https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html

[12] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). *Large-scale cluster management at Google with Borg*. Proceedings of the Tenth European Conference on Computer Systems. ACM.