Microservices and Netflix

Niamh Dunlea[1], Dominion Gbadamosi[1], Gabriela Lyko[1], Jai[1], and Kamil Jablonski[1]

University of Limerick

1 Introduction

An increasing number of businesses are adopting the microservices architecture as the paradigm chosen for their infrastructure. One of such companies is the leading streaming service provider Netflix. According to a report by Digital-Trends, Netflix was the top streaming service of 2023 with 260.28 million active subscriptions as of December 2023 [9]. Such a high demand for their services results in one outcome: lots of data! You must consider that Netflix must process 260.28 million account data alone. Then data regarding movies and tv shows suggestions, subscriber feedback, subscriber activity and feed predictions must be added to the equation. This poses the question: why has Netflix chosen the microservices architecture as their pattern of choice? Our essay will tackle this question, we will begin by discussing the microservices architecture to obtain good foundational knowledge on what exactly microservices are and how they operate. We will cover the architecture principles and its challenges. Furthermore, we will compare microservices to its predecessor, the monolithic architecture. This will bring us to the point in time when Netflix made the impactful decision to utilise the microservices architecture. We will discuss the migration process Netflix undertook and how they segmented their application into the microservices we are discussing. To build upon this, we will explore some external services integrated into Netflix that maintains their in-demand service streamlined. Finally, we will investigate Netflix's use of microservices at the granular level. This includes discussing how the software engineers have divided up the system into sub-systems that are able to tackle the flow of big data. We will explore concepts such as scalability, integrations, network traffic, load balancing, caching, and release management. These concepts quickly transform into issues if mishandled thus detailed discussion will be carried out.

2 Understanding Microservices Architecture

We initially leveraged the resources provided within our module to lay the groundwork into our research into microservice architecture. These materials offered insight into the fundamental principles and concepts inherent to microservices. We subsequently delved deeper into the topic of this architectural paradigm, employing the topics obtained from these resources as guiding beacons for a comprehensive investigation. Dave Cremins from Intel introduced us to the

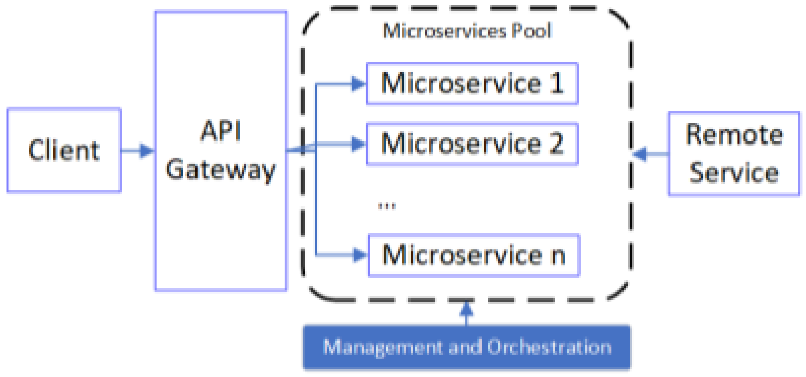


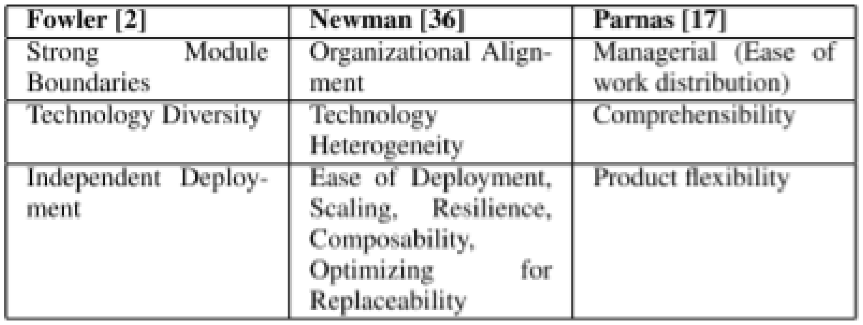
Fig. 1. Microservices Architecture [8]

evolution of this architecture, he mentioned that ”Microservices are an evolution of SOAL, we went from monoliths, to services, to microservices”. [17] In my exploration of the microservices architecture, a video by ByteByteGo, provided to us in the resources on January 29th, offered a crucial insight: ”This decomposes an application into a collection of small loosely coupled services. These services implement specific business capabilities, contain their own business model, and communicate with APIs.” [7]. Hence, the concept of loosely coupled services, alongside the delineation of specific business capabilities and models, as well as the integration with APIs, represent foundational principles within this architecture. These principles will be expounded upon in greater detail later in the essay. Furthermore, the content creator elaborated on practical applications and discussed how “Netflix uses microservices to handle everything from movie recommendations to billing. This architecture promotes modularisation of functionality. So, services can be developed, deployed, and scaled independently”. [7] Thus, the modularisation of functionality must also be added to the list of principles of this architecture. Finally, ByteByteGo discussed challenges associated with the microservices architecture: “It increases agility and allows companies like Netflix to innovate. The trade-off is to add a complexity managing inter-service communication and consistency.” [7]. To summarize, we composed a list of the obtained principles of the microservices architecture (and some of its challenges) from the provided resource and expanded on the topics using a research paper from the IEEE Access journal published in 2023. This paper ‘A survey on Microservices Architecture: Principles, Patterns and Migration Challenges’ provided us with in-depth analysis of the highlighted areas. Thus, here we provide explanations and foundational information to aid further discussion:

2.1 Principles

- Loosely Coupled Services: “When a microservice is designed, it is considered to have loose coupling, of such that it works independently. In the event of a failure, only this microservice will not be available and there will be no impact on the entire system”. There are several situations where such attribute proves to be useful i.e. functional testing, removal of redundant services, addition of services, debugging. This means the system is always online even in the events of a service being offline.
- Individual Business Capabilities: “Each microservice is developed to fulfil a specific business functionality and is not developed based on logical layers.” [8]. These microservices are developed with the loose coupling mentioned above and high functional cohesion. These attributes mean that these microservices interact with each other through its messaging system.
- APIs: “When microservices are designed and created, an API is generally used, which offers the necessary services so that the application can interact with the backend.” [8]. We can see this in Fig. 1 where the client (application) communicates with the relevant microservice(s). The remote service (backend) is called upon by the microservice(s) to complete the required functionality. This displays the flow of communication in the microservices architecture.
- Modularisation of Functionality: The core purpose of modularisation is to “allow modules to be reassembled and replaced without reassembly of the whole system”. [8] Thanks to the decomposition of the system into subsystems and their individual business capabilities, these microservices independently deployable which means any modification i.e. refactoring/ addition/removal can be performed without risk of knock-on effects to the whole system.
- Agility: “In Microservices, deployment is carried out in an agile way, relying on continuous delivery, reducing the time between the idea and the running software, with the benefit that organizations van respond quickly to market changes and introduce new feature faster.” [8] Thanks to the reduced scope that a developer must understand, the implementation, testing and evaluation can be carried out without high risk to the overall system. It must also be noted that due to the size of an individual service, these components are easier to comprehend as once again the whole scope of the system doesn’t need to be understood to work and possibly improve a service.

Additionally, in the guest lecture on the 23rd of April, Dave Cremins highlighted the most advantageous aspects of this architecture, he summarised ”The changes I make to a microservice, is only deployed there which is the greatest advantage of microservices. It’s cost effective” as well as ”You isolate areas into microservices. A microservice has a well defined API that other services can leverage. When you need to scale, you only scale independent services.” [17]



Fowler [2]	Newman [36]	Parnas [17]
Strong Module Boundaries	Organizational Alignment	Managerial (Ease of work distribution)
Technology Diversity	Technology Heterogeneity	Comprehensibility
Independent Deployment	Ease of Deployment, Scaling, Resilience, Composability, Optimizing for Replaceability	Product flexibility

Fig. 2. Principal benefits by acknowledged authors. [8]

2.2 Challenges

- Inter-Service Communication: Due to the nature of the architecture i.e. the increased number of services, there is a greater number of calls being made to services. This can be attributed to "causing communication delays between microservices and network overheads". [8] Delays in communication can prove to be costly and damaging to the reputation of the architecture thus solutions usually are sought at infrastructure or programming level, In terms of programming, developers may be inclined to refactor their code to enable asynchronous calls and as well parallel programming. At the infrastructure level, with the growing number of microservices and calls, a limitation of the infrastructure maybe be found, and costly upgrades may be needed to successfully integrate the growing communication needs.
- Interservice Consistency: This architecture "recommends that each microservice have its own database" [8]. This adds complexity to data storage as "it will be necessary to duplicate data in the different new databases created which will required. . . consistency of the data" [8]. Additional measures will have to be developed i.e. automated checks etc. to guarantee consistency, and this also should be regularly checked to always ensure consistency especially when a modification to the data has been made in a microservice.

2.3 Monolithic Architecture

Today businesses face a critical decision in choosing between monolithic and microservices architectures for application development. While monolithic architecture has been the standard, the era of the subscription-based model businesses has began to show the limitations of monolithic, leading many business to explore the benefits of microservices architecture. [18]

Monolithic architecture embodies a more traditional software development approach, where all application components are integrated directly in a single codebase and deployed as a single unit or entity. With its simplicity and suitability for rapid deployment it has been the standard for many years however it falls short in terms of flexibility and scalability as compared to microservice architecture as everyday applications grow in complexity and scope. [18]

With microservice architecture adopting more of a modular approach, with all its loosely coupled applications in the ecosystem and the ability of communicating between them via APIs. This architecture compared to monolithic brings about agility, scalability, and fault isolation, allowing businesses to scale more broadly with their individual coupled components to meet evolving demands of different sectors.

When comparing the two architecture patterns they have different use cases but there is a reason that the industry is shifting towards a more modular microservices approach and monolithic systems have long been declining in popularity among companies. This mainly equates to monolithics antique approach to a time where live service systems weren't as common and apps were deployed and done.

2.4 Main Comparisons

- Microservices: Excel in scalability, enabling independent scaling of individual services based on demand of sector/department.
- Monolithic: Scaling is more of an issue as the entire system will need to be scaled and replicated which is resource intensive and time consuming compared to microservice.
- Microservices: Allow quick and rapid updates to systems without disrupting or shutting down other services for maintenance.
- Monolithic: Doesn't work well with rapid updates and instead disturbs other systems when maintenance work is being done. Changing one component can domino into the whole system.

In summary, while monolithic architecture offers simplicity and speed of developing, microservices architecture provides the necessary scalability and flexibility that large complex on demand systems need nowadays.

2.5 Message Router Patterns

The concept of microservice architecture can also be compared to the architecture design of enterprise integration pattern message brokers. Message brokers aren't monolithic and are there to help tie complex systems together instead of just pure individual components. [19] With this comparison some shortcomings can be seen that should be avoided with microservice architecture.

Mainly message brokers usually rely on a "Central" broker that can communicate between components of a system [19], this can easily lead to speed issues as having so much information being passed through a central pipeline can lead to bottlenecks. This approach would still be considered as microservice but wouldn't be efficient due to one component doing so much heavy lifting in a system. This can be alleviated by having multiple "Central" brokers that can link different components together not needing all traffic to be funnelled through one tunnel.

This concept shows that when doing the microservice architecture its good to spread the system net wide and not have too many components relying on some central compound ones. Having your system be bottleneck free would be a major goal in a strong efficient system. These concepts offer a strong foundation on how Microservices should be used to further a system network.

3 How Netflix Uses Microservices

3.1 Netflix Microservice Architecture

Netflix has made its way into our lives by changing the very concept of what we consume through it. It, undoubtedly, provides the best services, giving us nothing but the right kind of experience. The way it has delivered these personalized services to millions of viewers, especially around the world, has truly been remarkable. Netflix's cloud practice started in 2009 with the cloud architect at that time, beginning with the migration of other applications such as content transmitting. Just by the year 2010, a new generation of users who could register their account after 2010, saving account information and watching all their favourite videos in one place, was introduced. By the end of 2011, Netflix finished migrating to a microservices architecture, microservices being the mark of a growing cloud computing paradigm. This shift so was quite traumatizing. Netflix was dependent on its traditional on-site (on-premises) servers; consequently, when introducing the cloud system, both versions were implemented in parallel for proper service continuity. It involved replicating large volumes of data to AWS data centres and there were many data management concerns such as managing the latency as most of the data centres were quite far away. During the time when the Netflix team faced several obstacles, including the extra load demand, instance failure, as well as performance bottlenecks. [14]

What fuels this impressive operation are special services working together and in a perfect harmony with container management platforms like Titus. Titus, with this power container, puts thousands of containers in play every day, operating like a perfectly functioning well-oiled machine. This epic tale of operations relies on containerization, ensuring scalability and efficiency, comparable to no other. With 3 million containers every week, we can see how big a user base is for Titus, knowing Netflix is committed to providing the utmost streaming experience. EVCache is another invaluable tool in the Netflix tech arsenal; it brings down the overall data retrieval speed through caching frequently accessed information. By caching the data inside the SSDs, Netflix can fine-tune the rapid data execution and the operations' stinginess. This innovative approach can give Netflix users an outstanding result and a cost-effective solution.

The architecture of Netflix is designed to make sure integrity and the endurance of services stand. Critical services run in isolation for uninterrupted user interaction, whereas stateless services take API requests with utmost availability. This strategic division shows the approach of unwavering commitment by Netflix towards providing a dependable and consistent experience to the user. [16]

Netflix's watchful guardian is Hystrix, a protective library that ensures proper access between services in an environment where failures may be possible. This script isolation makes it possible to respond quickly to failures or to monitor real-time recovery, and so on. Hystrix stands as a stalwart guardian, making sure that Netflix's services work smoothly even in the worst cases, with harsh winds.

At the forefront of Netflix's personalized user experience is a dynamic stream processing pipeline. This pipeline powers dynamic processing, collection, aggregation, and movement of microservice events in near real-time, all while managing business analytics and personalized content recommendations, fine-tuned to meet individual interests of every user. Integration in Kafka and Apache Chukwa reflects Netflix's commitment towards efficiently managing large amounts of data in a way that constructs an immersive viewing experience for every viewer. [15]

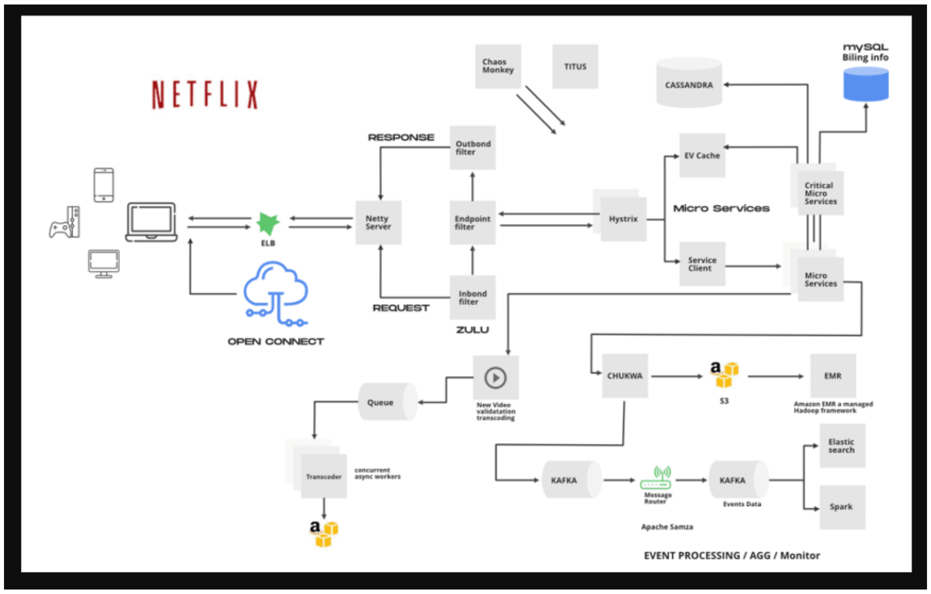


Fig. 3. Netflix Content Delivery Network [15]

3.2 Traffic Demand

With the high volumes of users accessing the service at any one point, it is critical that Netflix can handle high volumes of traffic through different strategies within the microservice architecture. To manage increases to traffic, Netflix uses a combination of strategies such as:

- Load balancing
- Scaling
- Caching

Load Balancing Managing the demand of services to the customers, using client-side load balancing is crucial for Netflix performance. Client-side load balancing means that the customers application is responsible for ensuring the appropriate service request is sent to the correct microservice. If the load balancer was at a central point within the system where all the requests had to come through and be disturbed it would cause a bottleneck. Having the load balancer client-side allows for a smoother and quicker response from the system for the customer's request and a lot more cost effective versus server- side load balancing [1]. For example, let's talk through a scenario of a user wanting to change their subscription. The user would navigate to their account settings on the Netflix application or website, and the client-side load balancing logic will discover the microservices responsible for handling subscriptions. The client-side load balancer will check to make sure this microservice is available and is not already overloaded with requests from other users. At this point, it will also take user specific information into account such as the users location as it wouldn't make sense for a request coming from a user in America to then be sent to the Europe specific microservice.

Netflix developed their own open-source service discovery tool called Eureka. This allowed for communication and coordination between microservices within the whole system. Eureka streamlines interactions with the users and services. With the client side load balancer, Eureka allows for information about servers and the ability for the instance to contact directly, speeding up the response time and avoiding unnecessary traffic within other microservices. This is also backed up by the use of servers cached on the client so in the event of an outage with a load balancer, the information is with the client and does not need to request the information new each time [2].

Scaling Netflix is so successful due to its ability to provide high-quality streaming experiences to a global audience. This can be attributed to their highly scalable infrastructure that can adapt to user demands. Netflix operated from a monolithic architecture, which became problematic as the user base grew [13].

The problem with operating from a monolithic architecture is that it forces all the functionalities into a single codebase, which can become difficult to maintain and update, especially as the feature set and user base expand [12]. Scaling a monolithic application often involves scaling everything at once, which is inefficient and expensive.

By adopting AWS as a microservices architecture, Netflix divided the platform into independent, self-contained services, and like what we learned from

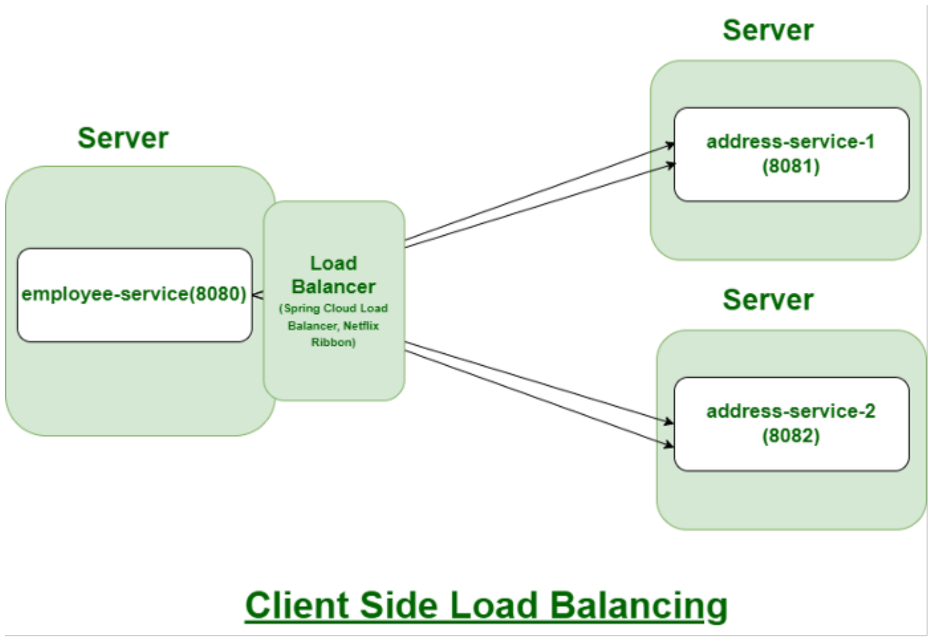


Fig. 4. Client Side Load Balancer [1].

Dr Andrew Le Gear, embracing cloud services is a great way to have maintainable architectures. Each service serves its purpose, e.g. recommendations, user authentication, or content delivery. Taking this modular approach offers many scalability advantages. For example:

Independent Scaling: With Microservices, Netflix can scale individual services up or down, depending on real-time demand. For instance, during peak viewing times, the video streaming service can be scaled up on AWS, while user account management may remain the same. This approach reduces costs and optimises resource allocation [13].

Flexibility and Agility: Microservices allow for increased speed in development and deployment cycles. Changes or updates to one service could have minimal impact on others, which facilitates quicker innovation. This increase in speed is crucial for a company like Netflix to stay ahead of its competitors in the streaming landscape [12]. Dave Cremins from Intel mentioned at his guest lecture that "Netflix is AWS biggest customer. Even with Amazon's AWS, they needed to figure out how to scale their application". [17]

Benefits of AWS for Scalability: AWS provides features like Auto Scaling, which automatically adjusts resources based on predefined metrics, eliminating the need for manual provisioning and ensuring that Netflix has the capacity it needs at any given moment [10].

The combination of AWS and microservices has been essential in Netflix's ability to scale efficiently and deliver seamless streaming to its millions of users.

While moving to a microservices architecture on AWS provided significant advantages, Netflix enhanced its scaling capabilities even further with Titus, its open-source container orchestration platform [11].

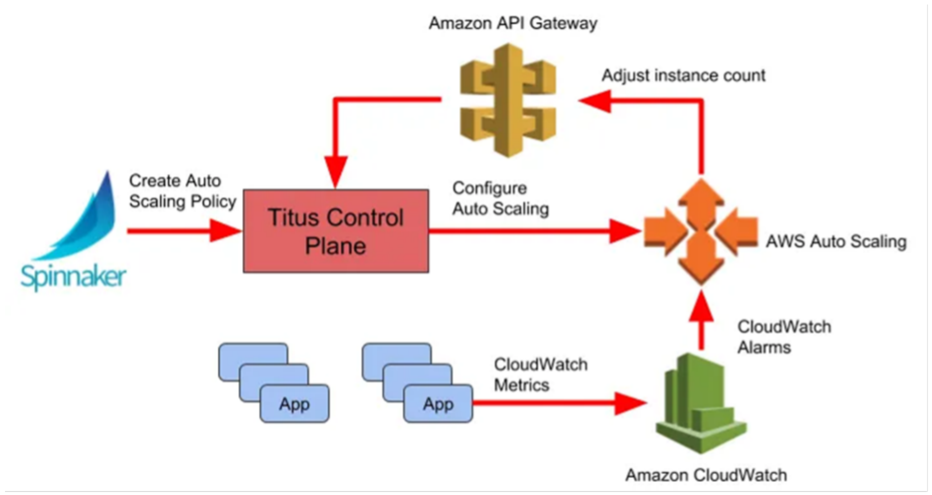


Fig. 5. Titus Auto Scaling workflow

Titus integrates seamlessly with Auto Scaling, which enables the automatic scaling of services in containers. This ensures resources are dynamically adjusted to meet fluctuating user demands (Netflix Technology Blog, 2019). Additionally, Titus utilizes AWS Auto Scaling policies and CloudWatch alarms to enable automated scaling decisions (Netflix Technology Blog, 2018).

Caching "Caching – the process of storing data in a cache, which is a temporary storage area that facilitates faster access to data with the goal of improving application and system performance" [3]. The use of caching has multiple benefits for both the users experience and also supports Netflix' ability to get information to the user as fast as possible. Scott Mansfield engineer at Netflix, mentions how they only have 90 seconds before a user will wait before moving onto something else. This means that if the user had to wait for something to load and it took longer than that, the likelihood of the user getting fed up and leaving the service is high. Netflix built their own in-memory caching system called EVCache to handle the huge traffic volumes and to serve content to the users with low latency [4]. Information that is used regularly, (e.g the users personalise homepage) will be retrieved from EVCache based on the user's previous behaviour and data on their account. This allows Netflix to have a faster response time while presenting to the user with recommendations or the ability to 'continue watching', without the need for querying databases for each request [5].

3.3 Release Management

With the high amount of customers it is vital that Netflix keep the service failure free and running 24/7. The ability to release new updates to the service without impacting customers is helped by the use of microservice architecture. Isolating each service means that if they are updating one service it is less likely to have an impact on any other service and therefore won't be noticed throughout the whole system.

Netflix also utilises a canary release deployment strategy when they roll out new versions. Canary releases within software development means that a smaller group of users will be exposed to the new version before it is rolled out to all users. This works by diverting traffic between the two versions [6]. This reduces risks from any issues or bugs within the new code as it would only be seen by this small cohort on the production system and allows for roll back scenario in the event the new version is a failure. Key metrics are reviewed in comparison between to the two versions and a decision would be made on whether all the traffic should now go to the new version or directed to the previous stable version.

4 Conclusion

As part of this paper, we reviewed the monolithic architecture and the need for a solution for organisation, as systems expanded and services demand increased.

Using microservices architecture by Netflix has revolutionised it's operation, allowing it to efficiently manage the huge amounts of data connected to their 260+ million subscribers. Moving from a monolithic to a microservice architecture, has provided advantages to Netflix such as scalability, flexibility, fault isolation and release management.

As well as benefiting the end customer with a reliable and high performing service, microservices support developers with release management during new deployment of separate services without impacting the whole system and allows for more options with scalability. Survey carried out on companies that use microservice architecture noted that "releases went from every 4-6 weeks to 3-4 times a day." allowing for more flexibility with maintenance and refactoring of code. [17]

At the start of this paper we asked "why has Netflix chosen the microservices architecture as their pattern of choice?". From research, it is clear to see the need for scalability, flexibility and maintainability was a driving factor for Netflix when selecting mircoservices. With the use of auto-scaling this offers Netflix the ability to future proof the business for years to come.

References

1. Geeks for geeks. 2024. 'Spring Cloud – Difference Between Client Side and Server Side Load Balancer', available: `https://www.geeksforgeeks.org/spring-cloud-difference-between-client-side-and-server-side-load-balancer/`. Last accessed 25th March 2024

2. Ranganathan, K. 2012. ‘Netflix Shares Cloud Load Balancing And Failover Tool: Eureka!’, Netflix Technology Blog available: https://netflixtechblog.com/netflix-shares-cloud-load-balancing-and-failover-tool-eureka-c10647ef95e5. Last accessed 25th March 2024
3. Sheldon, R. 2023. ‘Caching’, TechTarget, available: https://www.techtarget.com/whatis/definition/caching. Last accessed 25th March 2024
4. Strange Loop Conference. 2017. ‘Caching at Netflix: The Hidden Microservice” by Scott Mansfield, [video] available:https://www.youtube.com/watch?v=Rzdxgx3RC0Q. Last accessed 25th March 2024
5. Madappa, S. 2013. ‘Announcing EVCache: Distributed in-memory datastore for Cloud’, Netflix Technology Blog available: https://netflixtechblog.com/announcing-evcache-distributed-in-memory-datastore-for-cloud-c26a698c27f7. Last accessed 24th March 2024
6. Graff, M., Sanden, C. 2018. ‘Automated Canary Analysis at Netflix with Kayenta’, Netflix Technology Blog available: https://netflixtechblog.com/automated-canary-analysis-at-netflix-with-kayenta-3260bc7acc69. Last accessed 25th March 2024
7. ByteByteGo (2023) ‘Top 5 Most Used Architecture Patterns’, System Design Interview [video], available: https://youtu.be/f6zXyq4VPP8?si=VS-3krF_nDxejn5c. Last accessed 29 Jan 2024
8. Velepucha, V. and Flores. P. (2023) ‘A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges’, IEEE Access, Volume 11, 88339 – 88358, available: https://ieeexplore.ieee.org/document/10220070 Last accessed 25 Mar 2024
9. Digital Trends. (2024). The 10 most popular streaming services, ranked by subscriber count. [online] Available at: https://www.digitaltrends.com/home-theater/most-popular-streaming-services-by-subscribers/#:~:text=Netflix%3A%20260.28%20million. Last accessed 2 April 2024
10. Amazon Web Services. (2024) - Auto Scaling. https://docs.aws.amazon.com/autoscaling/ Last Accessed 30th March
11. Netflix Technology Blog (2018) Auto Scaling Production Services on Titus https://netflixtechblog.com/auto-scaling-production-services-on-titus-1f3cd49f5cd7
12. Richardson, C. (2019). Microservices Patterns: with Examples in Java. Shelter Island, New York: Manning Publications.
13. Netflix Technology Blog (2016) The architecture behind Netflix’s global streaming platform https://netflixtechblog.com/tagged/cloud-architecture
14. Venugopal, S. (2020). The Story of Netflix and Microservices. [online] GeeksforGeeks. https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/ Last accessed 17 March 2024
15. Olugbenga, E. (2021). Netflix System Design- Backend Architecture. [online] DEV Community. https://dev.to/gbengelebs/netflix-system-design-backend-architecture-10i3 Last accessed 29 March 2024
16. TechAhead. (n.d.). Let’s Decode Netflix System Design and Backend Architecture. [online] https://www.techaheadcorp.com/blog/netflix-system-design/. Last accessed 30 March 2024
17. Cremins, D. (2024) ‘Kubernetes in the age of Cloud Native’, CS4227: Software Architecture, 23 Apr, University of Limerick, unpublished.

18. Angela Davis (2023) Monolithic vs Microservices Architecture: Pros, Cons and Which to Choose [online] `https://www.openlegacy.com/blog/monolithic-application` Last accessed 19 Mar 2024
19. Hohpe, G. and Woolf, B. (2004) ‘Message Routing’, in Enterprise integration patterns: Designing, building, and deploying messaging solutions. 1st edn. Boston, Massachusetts: Addison Wesley, pp. 208–291.

”Introduction”, ”Understanding Microservices Architecture” , ”Principles”, ”Challenges”, ”Conclusion” G.L; ”Monolithic Architecture”,”Main Comparisons”, ”Message Router Patterns”, ”Conclusion” K.J; ”Netflix Microservice Architecture”, ”Conclusion” J; ”Traffic Demand”, ”Load Balancing”, ”Caching”, ”Release Management”, ”Conclusion”, Overleaf editing, Gantt Chart, N.D; ”Scaling”, ”Conclusion”, D.G;

All authors have read and agreed to the published version of the manuscript.