Disentangling the Monolith: Challenges and Benefits

Hannah O'Connor, Catherine Mooney, Ruairi McGrath and Senan McCague

Dublin City University,
School of Computing

hannah.oconnor26@mail.dcu.ie, catherine.mooney33@mail.dcu.ie,
ruairi.mcgrath33@mail.dcu.ie
senan.mccague2@mail.dcu.ie

Abstract

This paper investigates *Disentangling the Monolith*, by looking at its compelling challenges and associated benefits, through inspecting appropriate methodologies for such a transition and well-defined design patterns to be followed. It looks to real life industrial examples of world renowned organisations and developer's past experiences to form arguments for various use case types. Conventionally, the hyped *Microservices* has been the clear choice to migrate to for most. But through diving deep into the comparisons, contrasting highlights and challenges of its counterparts, this paper presents multiple other alternatives to be considered. Whilst sections of the paper will uncover the root cause for a move from *legacy* to modern, state-of-the-art distributed solutions, it contarily criticises if a need for change is present at all. That is, if it's nothing more than another fashionable trend within the software industry.

Keywords

Software, Monolith, Microservices, Macroservices, Modular Monolith, Miniservices

Introduction

In the beginning of this architectural evolution, software applications were solely monolithic based. Company's engineers would develop a central program which acted for all of the responsibilities and required just one deployment to ship to its end users. As time proceeded, many favoured modularization. This saw the introduction of **Service-Oriented Architecture** (SOA). As the internet's accessibility and availability grew, so did the customer reach of many within the software development industry. This commonality of the internet at a business-customer level meant that developers needed to match this rate of change. To enhance their software components, to handle a larger demographic, by essentially obtaining the ability to scale on demand, push to production at a faster pace and at lighter weight. As this need became an essential to stakeholders, microservices arose, seeing the disentangling of the once leading monolithic architecture.

With that overgrowing hype that follows microservices, it's evident that the software development industry is susceptible to trends. At times, it can be a thought-provoking challenge for many developers when judging if it is the next big movement for their software or if it's nothing more than a light change in their e-commerce applicatons's architectural designs from the heavyweight monolith. Despite the well-known monolith being a normality for numerous companies 10 years ago, their development teams must delicately access such a move, before falling carelessly into the trap of denigrating and risking the wrong decision for themselves and customers.

It is needless to say that all software companies are to adopt microservices. A software's architecture is a never ending responsibility, it's the foundation of a company's solution. But no architecture fits all - it's tailored to each individual and unique application. Regardless of which pattern a company follows, how it's managed, it's corresponding tech stack and overall system at play, it has a direct consequence on developers' happiness, eventually mirroring the productivity levels. The better the working environment, the happier the workers, and inversely the higher the quality of the code and product. But a change in architecture could be inevitable in such a competitive landscape.

1.Background

1a. What is a monolith application? Hannah

According to ancient greek, a monolithic was defined as a "*single stone*". Today, a monolithic-based application is referred to frequently in metaphorical terms, being a one deployable unit of software approach. In this client-server based architecture, a monolithic contains numerous entities within the one system, across various domains. These can include the user interface (UI), the business logic and persisted data storage. Not to forget the addition of its modules, components, frameworks and libraries too.

Within Sam Newman's book "*Monolith to Microservices*", it discusses the various derivations of the generic monolith architecture model. Those various classifications being: the single-process system, the distributed monolith, and third-party black-box systems. Each needing their own tailored treatment when it comes to the latter dissolution. The most common, single-process system, similarly to

modular monolith without the separation of modules, refers to a single deployed unit. The distributed monolith, which can be addressed as a microservice transition *gone wrong* [1], defines a system with several services which still required to be deployed together. Lastly, third-party black-box systems can be used to characterise other developer's software. If proprietary, it makes it unchangeable. [2]

1b. What are the advantages of a monolithic architecture? Catherine

In recent times, Monolithic architectural systems are becoming increasingly known as being *outdated*. However, this view is incorrect, as they can be majorly beneficial, depending on their usage. As monoliths are so simple to implement, it makes sense to consider them as an asset, depending on the systems complexity. The main benefits are performance related, as testing, building and deploying a single unit tend to be unambiguous, as it does not come with the complexities of dealing with multiple individual systems. In addition, there is a lot less operational overhead. Monoliths are extremely beneficial if they are used within a small team, rather than a large organisation. As all code is built in a single application, data is stored in the one place, making it easily accessible. Furthermore, it can avoid the common difficulties faced with communicating between various components in a distributed system, as components are called directly. [3, 4]

2. Motivation

3.1 The motivation for change

It can be argued that the monolithics time has been up for quite some while. Back in 1997, the "*big ball of mud*" nickname was introduced within a talk by Brian Foote and Joseph Yoder, which was quickly associated with the well known monolith across the industry. [5] This has established the monolith as something to avoid. Whenever coming across the monolith architecture, it was commonly paired with the word *legacy*. [2]

Many of the monolith's faults can push developers to change. Following years worth of development and growth, the disadvantages of remaining with an overgrown monolithic architecture can tower its once redeeming qualities. Based on the common attributes of the monolith, many developers work on the encapsulation of functions within their application's self-contained software. Conversely creating a tightly coupled architecture. Given that rigid dependability, any single bug found across the many components can be the potential source of an application failure. According to the **Constantine Law** "*A structure is stable if cohesion is high, and coupling is low*" [6]. That high coupling element, can essentially place a limitation on the future large-scaled changes, often snowballing into an unmaintainable codebase.

The ability to scale on demand for a project is a crucial aspect in today's competitive market. Completed studies published in journal "*How To Make Your Application Scale*" refer to six key areas when scaling an application. Those include: Distribution, Non-Uniform Scaling, Portability, Elasticity, Availability, Robustness. [7] Where a microservices type architecture excels within them, a monolith can struggle and sometimes fail. Also, there's an additional overhead with monolith when its modules differentiate within their resource requirements. Consequently, hindering chances of reaching scalability. Additionally, it brings about more difficulty when embracing new technologies to it's embedded stack. Adopting a new framework could be an entire application rewrite. [8]

Continuous deployment can be another negativity for a monolithic-based development team. While automating the releasing process may be a good idea, due to the underlying nature seen across monolithic architectures, updating one component means an update to all. Not to mention higher chances of experiencing a single error to put a brake on the entire process, causing frequent restarts and hindered productivity. That time taken to constantly work towards fixing bugs and temporary patching to prioritise delivery over quality sees long standing effects. Even a new release is pushed, the possibility of breaking within production hovers. [8]

Given no complications, the decision to refactor or perhaps re-architect could be best decided as late as possible. Effectively giving the chance for developers to thoroughly understand their codebase and its overall corresponding system inside-out. Before reaching domain expertise, it can lead to a more complex future implementation of a distributed system.

1d. Is change necessary?

No rule exists where a change in architecture must be done or that development companies must follow tech trends. Starting with a monolith architecture is just as valid a choice as microservices. Difficulties lie in assessing which architecture is the best in the early development days. As discussed within the *Shopify Case Study*, it referenced Martin Fowler's Design Hypothesis where, in the early stages of software development, applications can be shipped very rapidly with little to no consideration to design. A good time to invest is when the speed of adding decreases. Despite their decision, one of Shopify's senior engineers said "*I would actually still recommend that new companies and new products start off with a monolith*".[9] Indicating that the delivery of service and customer satisfaction are the primary goals for the beginning of any software project. The details surrounding how it's delivered and what it's delivered on should come later.

Monoliths can be a very conventional way to start. Given their simplistic nature, developers can code and push to production at faster rates, without much consideration in architectural design plans. Within Martin Fowler's monolith-first strategy, he argued *"You should build a new application as a monolith initially, even if you think it's likely that it will benefit from a microservices architecture later on."*.[10] However, Stefan Tilkov later contradicted Fowler's point stating that "*you shouldn't start with a monolith, that is, if your goal is a successful microservice architecture*". [10] Referring to Simon Brown when he said *"If you can't build a monolith, what makes you think microservices are the answer?"*. [11]

The **First Law of Distributed Objects** declares that developers shouldn't distribute their system into a microservices-based alternative unless it needs it. [12] If a company owns a well, or perhaps only partially, modularized and structured monolith, then it may never need a time to make the move to microservices. Referring to Fowler again, *"You shouldn't introduce the complexity of additional distribution into your system if you don't have a very good reason for doing so."* [13]. Many well-known modern applications and online providers started with the conventional monolith. **Amazon.com**, for instance, back in 2001 was an architectural monolith. Despite being multi-tiered, the various components across its tiers were tightly coupled together. [14] Amazon AWS Senior Manager, Rob Brigham, and spokesperson for the retail site addressed the experience through advising other companies with the following:

"*Now, a lot of startups, and even projects inside of big companies, start out this way. They take a monolith-first approach, because it's very quick, to get moving quickly. But over time, as that project matures, as you add more developers on it, as it grows and the code base gets larger and the architecture gets more complex, that monolith is going to add overhead into your process, and that software development life cycle is going to begin to slow down*." [14]

2 Methodology on changing from monolithic architecture (Hannah)

There are several pre-stages that need to be planned before diving head first in such a transition. Some might include: planning, conducting feasibility, lining out and analysing business requirements, various business domains, the actual architectural design and definition of the distributed system, the functionality of each service and the development plans for the Continuous Integration and Delivery pipelines. But as great as a microservices-based architecture can be against a legacy monolith, it's not always the silver lining to success. Not all monoliths require decomposition in the first place.

In Sam Newman's book '*Monolith to Microservices',* there's 3 key questions to ask the organisation as whole when it comes to the considerations of breaking the monolith to, primarily, microservices. Theses are:

What are you hoping to achieve?
Have you considered alternatives to microservies?
How will you know if the transition is working? [2]

It's important to know that making a change like this doesn't just link to the architecture but the culture too. To see results, it requires strong correlations between the organisation's structure and their architectural knowledge. Meaning the teams involved must adapt to the system. **DevOps** is another tech term classified with microservices. In their simplest terms, both are based on the ideology of achieving greater agility and operational efficiency. With the modularity of microservices, it means that multiple DevOps teams can synchronize the building and deployment of their microservices, making the perfect team. Although, a DevOps adoption would not be necessary, it can often be the missing piece to a prosperous transition.

When it comes to designing a microservices architecture, Eric Evans's **Domain Driven Design** (DDD) can be adopted to kickstart the decomposition process. The idea behind this design pattern when alongside microservices is that it should be guided by domain boundaries. With each service or component encapsulating discreet business logic. [15]

There's many other existing architectural patterns which can be utilised when designing the transition's strategy. The Martin L. Abbott and Michael T. Fisher book '*The Art of Scalability*' discusses the concept of the **Scale Cube.** It discusses how infinite scaling can be achieved through an X,Y or Z-axis. Traditionally with microservices we see a lot of X-axis scaling, horizontal scaling, the running of N instances of a cloned application. With the general microservices architecture following along the Y-axis, the breaking of the application into various components and services. Z-axis approach focuses on the segregation of data across different servers [16]. Cockburn's Ports and adapters is another efficient approach. It's defined as:

'*Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.'*

It tends to be used for building reusable services. It implements the bounded context

pattern, designing for the decoupling of the application logic from inputs and outputs. [17]

There's many approaches to take with a disentangling strategy. An organisation could select to either refactor or rewrite their application. According to IBM, there's a *"four-phase roadmap to transform your application from a monolith to a set of microservices''.* This entails starting with macroservices, splitting into Miniservices, eventually achieving Microservices and then finally moving to a scaled out Microservices approach. [18]

As discussed in the latter Section 8, *Shopify and Modular Monolith*, the journey from monolith to microservices transition could be less complicated through first achieving a modular-monolith. Giving Shopify's success, that approach could mean improved chances of correctly defined component boundaries before addressing further complexity of a distributed system.

As mentioned, one of the first passable approaches is to completely rewrite. But it impractically depends on funding,engineering power and hours, more than its counterparts. It can't be marked as completed until both the new and original system are matched. Causing the engineers to be blindsided through the development process. This '*big bang*' approach of coding from scratch can be far too demanding of a task. As mostly it ends unsuccessfully: '*The only thing a Big Bang rewrite guarantees is a Big Bang!*' [19]

Alternatively, development teams could instead take the task of refactoring to incrementally achieve microservices. But neither this or the previous strategy discussed can be set in stone as being the right move for each unique monolith application. Various aspects of the codebase will need consideration like: How is it built? How is it packaged? How does the code function? How does the application interact with back-end data sources? How are those data sources structures? [20]

Whilst migration is ongoing, some services could exist within the new premise and others can be left waiting to be dealt with. Within Michael Feathers Working *Effectively with Legacy Code*, he defined the concept of a *seam*. He uses this to refer to isolated code that can be worked on without impacting other parts. Eventually those *seams* can act as the future service boundaries.[21] This idea of a hybrid architecture gives the ability to gradually build the new architecture while in conjunction with the original. However, this method doesn't necessarily mean less work. It requires identification of candidate components to be transformed, through accessing which, when rearchited, would yield greatest turnover. Yet, a hybrid approach is not an ideal permanent situation. [22]

Refactoring can too be a messy process, but the end result gives way to higher likelihoods of success. This route depends more on developers' knowledge and thoroughly rooted understanding of their codebase. Also, it's nearly impossible to do at once. Hence, it must be incrementally conducted, whilst running in conjunction to the monolith. The microservices application gradually grows as the monolith minimizes. Refactoring until the application is entirely microservices based. This approach can be referred to as the **Strangler Application**. [19] Those strategies to achieve and complete refactoring are: *Implement New Functionalities as Services*, *Split the Back-end from Front-end* and to *Extract Services*. [20]

The first strategy, in simplest terms, is stop adding to the monolith. So when a new business requirement is presented, it shouldn't be developed within the monolith. Instead, new additions will aid in working towards the migration completion through placing that new code within its own

service. It still doesn't address the actual monolith itself. Hence, more strategies are required to guide its complete disentangling.

Next is the *Split the Back-end from Front-end* strategy. Typical enterprise-level applications will be made up of a presentation, business and data access layer. With this approach, it works to separate presentation from the two lower levels, creating two separate applications. But neither will automatically switch from monolith. Despite enabling independence for UI and back-end developers, additional strategies are needed for the treatment of the monolithic code-base. Lastly, *Extract Services* focuses primarily on extracting modules from the monolith based on unique business functionality, using it as the basis for each to-be microservice.

Rather heading straight for microservices, the organisation could divert, refactoring into macroservices before a resulting microservices architecture. On the other hand, according to **Allen Holub,** usually when most organisations attempt to achieve microservices, they end with macroservices instead, through not breaking down the monolith enough, ultimately leading to latter work. [23] Alternatively, an organisation could take a different route from microservices altogether. Instead, one could transform the application through splitting the monolith into smaller serverless functions. This approach takes aways the need for infrastructural management as instead it requires code to be uploaded to a cloud provider's compute service such as **AWS Lambda**. It works by breaking the monolith into various functions, with a third-party cloud service provider, like AWS, taking full responsibility for the servers the application. Of course both Serverless and microservices can very possibly be integrated as one to achieve a one powerful architecture.[24]

Another game changer is *Self-Contained Systems (SCS)*, which are conceptually close to microservices. Both resulting in distributed systems and similar ownership of services policies, it too is recommended for a single SCS to be claimed by one team. Rather with SCS, it breaks the monolith into coarse-grained, autonomous, replaceable web applications. [25] Like microservices, it's ideal to implement a bounded context first and use those contexts as a guide for dissolving the monolith. While discussing microservices, it highlighted a key starting point to be identifying and extracting modules from their code base. Instead of solely extracting the backend, SCS concept is based on consisting of both the backend and corresponding UI. Essentially gaining a group of UI components and their matched microservies. [26]

With data management, strategies need to be in place, regardless of the goal architecture, in order to successfully migrate data. When it comes to it, there's three main routes. First, start with a new domain model with new storage. Thus, starting from scratch with no data. Next, start with a new domain model but instead, migrate the original application data before it's pushed to production. Lastly, teams may have two running systems side-by-side, they could be separated completely or t have separate data but small amounts of synchronization. [27] **Stripe** have designed a *four phase data migration strategy* which focuses incrementally migrating applications that integrate through the database, while all the systems under change need to run continuously.

For the decentralization of the database for microservices, focus needs to be placed on the service decoupling, straying from the original single shared database. Keeping align with the microservices pattern, the best practice is to allow each individual service to manage a separate database store or instance with its own domain data. Otherwise independently deployment and scaling isn't possible. In

brief, microservices need to incorporate DDD when it comes to this type of planning, the dividing up of a domain into multiple bounded contexts. It follows the guide of designing models first. [28] With each database being owned by only one service. A common design to allow for consistency among multiple is the **Saga Pattern**. [29]

The company structure, following the breaking into a distributed system, it changes the way in which each engineering team must discern the application's stack. According to the **Conway Law,** it states:

> *"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure".* [30]

It discusses the best practices on adopting a software structure that will be closely aligned to the organisation's communication. Such as DevOps with Microservices. As the structure of one slowly changes then small movements should be taking in the other. This law focuses on achieving a higher productivity level through having created a system that's built around the core concept of coordination and communication. [33] This move aspect is as important as any other as mostly the motivation surrounds teams autonomy and development speeds. Better delivery of services could be seen with both a distributed system and team.

3. (Catherine)

3.1 Accessing if Microservices is the right choice?

Over recent years, the Microservice architecture (MSA) has become increasingly topical amongst software companies. According to a survey conducted, 68% of companies are either using or investigating microservices. Yet, the popularity in its selections across the industry doesn't necessarily mean it will be a good fit for all organisations. It's a large scale project within itself that could take up to a year to successfully complete, with the associated depth and complexity being directly proportionate to the original applications' size and the engineering hours at hand.

MSA is a means of developing a software system that's built up of a series of small services. Each service communicates with each other using APIs and Rest interfaces, such as HTTP. The main goal of MSA is simplicity, it allows for businesses to split up their workload into units, which are then self-responsible. This eliminates having one team build, test and deploy an entire system, allowing for a smaller teams to work on individual tasks. Meaning each microservice is directly responsible for their own data store, while in a monolith there is a single database. Essentially, if implemented correctly, a service should not break if another does.[31,32,33]

While migrating to microservices can be complex, by decomposing the system into singular services, the system becomes much simpler to understand. Additionally, the services are much easier to maintain and faster to develop. Once migration to microservices is successful, the system can now use **polyglot programming** in order to allow teams to work efficiently. This is beneficial as it reduces the cost of finding specific developers that know the system's technologies, as with MSA developers can choose the technologies best suited for their specific project. However, this makes it difficult for

developers outside of a team to change the microservice as they mightn't have any knowledge of the technology used. [34, 35]

While there's many benefits in the migration to microservices, there's disadvantages too. Deciding which parts of the system to split up can cause difficulties in both deployment and operations. Companies often make the mistake of having microservices which directly depend on other services. In turn, causing more problems with additional cost, as rebuilding services leads to unnecessary redeployment.[36]

When testing microservices, it can be a lot more granular in comparison with a monolith. A team can run unit tests on each microservice, which focuses on the smallest testable part of software. This ensures that the system is working down to a much finer detail than in a monolithic architecture. But teams need to run integration tests to guarantee the system is working coherently with regards to the individual units, including their datastores. Integration testing can be quite costly and error-prone, as it is quite complex and timely. [37]

2.1 Alternatives to Microservices (Catherine)

The mistake companies often make is moving to a MSA expecting it to be easier to maintain. Often it's too big of a jump to move to MSA as moving away from object-orientated monoliths can cause complications. Sometimes MSA doesn't suit smaller products, while modularization does. Modular monoliths are an example of a SOA. Modular architecture has many benefits when companies don't have too many developers or a large code base. By using encapsulated components, a modular architecture can provide defined interfaces and encapsulated data [38]. Shopify is a well known example of a software company which saw great success after adopting the Monolith Modular architecture as an alternative to the microservice decomposition.

While each module in a modularized monolith is seen to be independent, it cannot be completely, so some integration with other system modules is needed. Each service can communicate with each other using a standard business processing layer. [39]

A company could decide, instead of using a high granularity of very small services, to split their system into a smaller number of services. Gaining many of the microservices benefits, without the many complexities. However, along with a monolithic architecture, adding functionality generally requires the developer to have an understanding of the entire system. This is due to the fact that services, within a macroservice architecture, also rely on each other.

Alternatively, there's the miniservice architecture. Miniservices are based around one single function. Allows teams to be in charge of multiple miniservices, instead of one single microservice. This ensures easier organisation within companies as teams can be responsible for related services, allowing for coordination with regards to databases and shared libraries [40]. As miniservices are based solely on a single function, it's more cost efficient, which, when being run, load up multiple functions at once. However, they are not always serverless, meaning services are running the entire time. While this may reduce that cost efficiency, they tend to be much more comprehensible than microservices. Another major difference between the two is that while it can share data between

services, microservices cannot. This is one of the most expensive aspects of the MSA. Shared data reduces the complexities of combining data from multiple services. [41]

Self Contained Systems (SCS) have many similarities to MSA, where the monolith is split up into separate independent functions and hyperlinks are used to navigate between systems. Additionally, they tend to be more precisely defined than that of a MSA. With each system being asynchronous, an SCS should work even if others are offline, therefore minimizing failure risks. Along with microservices, SCS allows for the use of polyglot programming. As systems are so loosely coupled, sharing no business code, they can be easily replaced. Alongside that, they also share no databases, allowing for flexible customization. Often SCSs become too big, so organisations use a SCS as an intermediary step before migrating to an MSA. [42, 43, 44]

2.2 Challenges of transitioning from a monolithic architecture

Despite transitions not being easy paths, with the initial issues that grow with the monolith, it can make an architectural change far more appealing. Arising challenges both contexts of organization and technical are equally important when it comes to being dealt with. As conflicts between the two will cause worse problems.

When it comes to the technical side of things, managing effective communication following the disentangling is prime when facing future challenges. Between the microservices, communication can be a lengthy process when trying to get right. If too finely grained, it can lead to deeper latter issues. In terms of the overall business process, cross-team communication goes hand in hand with it. Prior to the move, many organizations would find monolith embodying a restrictive quality when it came to engineering team sizes and their organization. In contrast to the new structure that requires it to be put right among the business' employees. It means individual teams must be made responsible for the upkeep of their services. This includes developing, testing, deploying and monitoring whilst in production. In contrast to the monolith where application runs as one, depending on a great deal of cross-team synchronization in terms of code conflicts.

A higher operational complexity will need to be kept in mind when it comes to how microservices are managed. This is where '*Microservices tax*' comes into the picture, referring to the extra cost of managing, monitoring, testing and debugging a distributed microservices system in comparison to the simplicit monolith architecture. But there is a solution. The network in between the services. Their communication can be made monitorable through that very network as a simple work around. [45]

3. Monolithic Transition Case Studies

3.1 Shopify Approach using Modular Monolith - Hannah

Shopify is a subscription-based software offering international customers their e-commerce platform solution. Today, they've surpassed 1,000,000 merchants with approximately 4000 employees. [27]. However, not too long ago, Shopify was a giant monolithic application. Kirsten Westeinde, Senior Developer at Shopify said [45] "*While it is no longer the best solution for us at shopify, it was for a*

very long time.". Within her talk at the *Shopify Unite Track (2019)*, Kirsten explains how at the beginning their priority was shipping the product, with all its necessary functionality in place, to customers, before focusing on the achieving of a perfect design. As time passed, the shopify developers eventually reached a point where it became increasingly difficult to incrementally deploy new code. They came to the realisation that they needed to invest within their design. [48] This has been a common occurrence in many monolith based projects. Martin Fowler had referred to this experience within his **Design Stamina Hypothesis**, classifying the moment as "*crossing the design payoff line*". [49]

The evolution of their software grew from when the symptoms they experienced highly indicated that they'd reached that point. Being that Shopify is one of the largest Ruby on Rails codebases in its industry, [50] the nature of the Ruby code meant it was globally accessible. This means that it has the ability to be called from anywhere without the explicit need of depending on it. Hence, being a good match for a monolith. [51]

In the hopes of striving to seek a sustainable architecture, Shopify Developers explored the trending microservices as a passable transitional path. But that contender was soon ruled out as the complexity that comes with a distributed system inhibited their wish to maintain a straight-forward single deployable unit. With their case, it displayed an illustration of how companies must evolve the software alongside it's ever-growing customers, and their requirements . Kirstien discussed Shopify's experience by saying:

"We realised all the things we liked about our monolith, were a result of the code living in and being deployed to one place. And all the issues we were experiencing were a direct result of a lack of boundaries between distinct functionality in our code"

Opposed to microservices, they opted for the Modular Monolith, which they felt comprised the best qualities of both the Monoliths and Microservices without the complexities. The end goal was defined as to "*increase the modularity without increasing the number of deployment units*". With a modular monolith, all the code that powers the applications lives in the same codebase and deployed together but with strict enforcements on the boundaries in between the different domains. The project discussions started back in early 2016 before making the decision to go ahead with "*Break-Core-Up-Into-Multiple-Pieces*" in late 2017 following the conduction of a company based survey, later becoming "*Componentization*", Shopify's personalized name for their *implementation of a modular monolith* project. It wasn't until 2019 the project completed.

The transitions goals were defined as: Reorganizing the code, Isolating code dependencies and Enforcing domain boundaries.[52]. When reorganizing their codebase, before it had followed the default Ruby on Rails directory structure, which would've been established by the helper script at the creation time of the application.[53] To accommodate the change, they refactored in order to gain modularity through the recreation of their components to model real world concepts, like billing, shipping, etc. What this movement did was essentially re-establishing individual applications within their own logical namespace. In turn, this creation of divisions in various aspects of the central code base, makes life much easier for their engineers when it comes to future development and their understanding.

Next came isolating the dependencies. This technical debt included the decoupling of components across domain boundaries. To achieve this, each business domain owner took responsibility for their component, taking advantage of their expertise for a successful decouplement. An overall componentization team also existed, to oversee each individual development teams work and to also provide default patterns to follow and essential tools to monitor violations rules.

Last step was enforcing boundaries. To improve the clarity in their architecture, following the isolation process, their development plan was to strictly grant component access to only those in which it explicitly depended on. As a result, through extracting code fragments, it has been made easier to suit any additional requirements in their foreseeable future.

3.2 Netflix Approach using Microservices - Catherine

Netflix was one of the first to make the monolith to cloud-based microservice architecture transition, allowing them to test, build and deploy independent global services without affecting others within the system. With over 150 million users in over 190 countries, and an estimated 37% of the world's internet users using their service, there's no doubt of the need for a large-scale network of servers. To accommodate it, they adopted Amazon Web Services (AWS) as their primary cloud provider alongside the establishment of their own local Internet Service Provider (ISP) data centers, Open Connect, to deliver their video content across the world in various regions, to provide for their global end users' devices. [54,55,56] Netflix spokesperson Joris Evers said:

"*The best way to express it is that everything you see on Netflix up until the play button is on AWS, the actual video is delivered through our content delivery network (CDN)*" [57]

Netflix started out in 1998, as an online DVD rental service, with its main competitor being Blockbusters. As the years went on, and online streaming services arose, Netflix soon broadened its services, to a subscription based site, straying away from individual DVD rentals. Subscribers could rent as many DVDs as they liked, with no additional fees associated. This saw drastic increases, with users doubling from 300,000 in 2000 to 600,00 just two years later. By 2007, Netflix became what is now known as worldwide, online streaming service. [57]

It wasn't until 2008, when a database corruption forced Netflixs shipping of DVDs to a halt, for three days, it became apparent at that moment that their monolithic architecture would not be sufficient to handle the capacity and demand for future services. In 2009, the Director of Web Engineering and then Cloud Architect, Adrian Cockcroft, revolutionised Netflix through introducing a microservices architectural solution. By moving from a single autonomous unit, Netflix could now have multiple small teams working on 100s of microservices, rather 100 engineers working on the same base. This transition now meant that Netflix could handle the exponential growth, as the company could no longer build data centers fast enough to keep up with the fast paced changes. With AWS they could now scale their databases in a much quicker manner. [55]

As the move required a complete remodeling of the entire systems architecture, there were some expected issues. It took Netflix a full 7 years to migrate the entire system to microservices, as they did not want to run into grave difficulty given a hasty transition. There were many costs that factored when transitioning, as the company was one of the first to deal with microservices, the complexities

were a strain on the majority of teams. This learning curve was mainly brought on by the Operations Engineering team within Netlflix decided to use Chaos Engineering as a way to tackle common issues when moving from data center to cloud.

With over 30 engineering teams working on seperate services, Netflix designed a library, Hystrix, in order for services to communicate with one another. According to Netflix:
"*Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services, and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.*"

The following code snippet from the Netflix/Hystrix github, samples how the HystrixCommand wraps the code to be isolated within the run method.

```
public class CommandHelloWorld extends HystrixCommand<String> {
    private final String name;
    public CommandHelloWorld(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }
    @Override
    protected String run() {
        return "Hello " + name + "!";
    }
}
```

The Following is an example of how the previous command could be used:

```
String s = new CommandHelloWorld("Bob").execute();
Future<String> s = new CommandHelloWorld("Bob").queue();
Observable<String> s = new CommandHelloWorld("Bob").observe();
```

[58]

Conclusion

This paper has investigated disentangling the monolith, examining individual challenges and benefits leading to it while briefly discussing its aftermath too. It's clear that no architectural approach is perfect and the dissolution of a company's current withstanding monolithic-based architecture is nor always necessary or even beneficial. This paper uncovered that Microservices is not only the option in the market, despite its high circulating trends. Many success stories can be seen with the legacy monolith, as concluded in the *Shopify and Modular Monolith* case study, just as frequently witnessed with an ecosystem of microservices. Our research has concluded that, generally as the monolith grows in size, it grows in its complexity and it's difficulty to be managed. Many companies will rely greatly while others become quite susceptible to move away and transition to a more modern approach for the architectural design. Commonly, the alternative is a more distributed and modularized system. While each monolithic application is unique, this paper highlights generic migration patterns, designs and resulting structures for a clear-cut disentanglement.

References

[1] Etheredge, J.(2018). You're Not Actually Building Microservices. SimpleThread. [Online]. Available: https://www.simplethread.com/youre-not-actually-building-microservices/
[accessed 3 March 2020]
[2] Newman S. (2019). Monolith to Microservices. 1st ed. O'Reilly Media, Inc. [Book] [accessed 1 March 2020]
[3] Lumetta, J.(2018). Monolith vs microservices: which architecture is right for your team?. FreeCodeCamp. [Online]. Available: https://www.freecodecamp.org/news/monolith-vs-microservices-which-architecture-is-right-for-your-team-bb840319d531/
[accessed 3 March 2020]
[4] Westeinde,K. (2019). Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity. Shopify Partners. [Online]. Available: https://www.shopify.com/partners/blog/monolith-software [accessed 3 March 2020]
[5] Brian Foote and Joseph Yoder, Big Ball of Mud. Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97) Monticello, Illinois, September 1997
[6] http://www.principles-wiki.net/principles:constantine_s_law
[7] Martin Fowler.(2015). Monolith First .[Online]
Available at: https://martinfowler.com/bliki/MonolithFirst.html [Accessed Feb 2020]
[8] Dragon, Lanese ,Larsen, Mazzara, Mustafin, Safina. (2017) Microservices: How To Make Your Application Scale. Arxiv Cornell University.
[9] Fowler, M. (2014). Microservices and the First Law of Distributed Objects. [Online]. Available: https://www.martinfowler.com/articles/distributed-objects-microservices.html
[Accessed March 2020]
[10] Martin Fowler.(2015). Monolith First .[Online]
Available at: https://martinfowler.com/bliki/MonolithFirst.html [Accessed Feb 2020]
[11] https://martinfowler.com/articles/dont-start-monolith.html
[11] M. Fulton III, S. (2015). What Led Amazon to its Own Microservices Architecture. [Online]. Available: https://thenewstack.io/led-amazon-microservices-architecture/
[Accessed Feb 2020]
[12]Evans E. (2003). Domain Driven Design: Tackling Complexity Software. Addision-Wesley Professional. [Book]
[accessed 1 March 2020]
[13] McGlothin, R. 2018. The Scale Cube. AKF Partners, [Online]
Available at: https://akfpartners.com/techblog/2008/05/08/splitting-applications-or-services-for-scale/
[Accessed 24 Feb 2020]
[14] Sciforce. 2020. Another story about microservices: Hexagonal Architecture. Medium, [Online] Available at: https://medium.com/sciforce/another-story-about-microservices-hexagonal-architecture-23db93fa52a2
[15] Brown, K. Refactoring application code to microservices. IBM, [Online],
Available at: https://www.ibm.com/garage/method/practices/code/refactor-to-microservices/ [accessed 22 February 2020]
[16]Shoup, R. 2014. Evolutionary Architecture. [Online].
Available at: https://randyshoup.silvrback.com/evolutionary-architecture
[Accessed 26 Feb 2020]
[17]Jake Lumetta. These are the most effective microservice testing strategies, according to the experts https://www.freecodecamp.org/news/these-are-the-most-effective-microservice-testing-strategies-according-to-the-experts-6fb584f2edde/ 2018
[18]Feathers, M.(2004). Working Effectively with Legacy Code. 1st ed.Prentice Hall.[Book]
[19]Koptyev, S. (2019). Why a Microservices Hybrid Model Is What You Probably Need Instead. The News Stack. [Online]
Available at: https://thenewstack.io/why-a-microservices-hybrid-model-is-what-you-probably-need-instead/ [accessed 1 March 2020]
[20] O'Reilly Software Architecture Conference. The three-headed dog: Architecture, process, structure - Allen Holub. (2019). Berlin, Germany [Online]

Available at: https://learning.oreilly.com/videos/oreilly-software-architecture/9781492050728/9781492050728-video328592 [Accessed 1 March 2020]
[21] Anastasia D. (2019). Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless?. Ruby Garage. [Online]. Available at: https://rubygarage.org/blog/monolith-soa-microservices-serverless [Accessed 28 Feb 2020]
[22] Wolff, E. (2017).Self Contained Systems (SCS): Microservices Done Right. InfoQ. [Online]. Available at: https://www.infoq.com/articles/scs-microservices-done-right/ [Accessed 28 Feb 2020]
[23] Matyashovskyy, T. 2017. Migration to Microservices: Lessons Learned. DZone, [Online] Available at: https://dzone.com/articles/migration-to-microservices-lessons-learned [Accessed 19 February 2020]
[24] Behara, S. 2018. Breaking the Monolithic Database in Your Microservices Architecture. DZone, [Online]. Available at: https://dzone.com/articles/breaking-the-monolithic-database-in-your-microserv
[25] Richardson, C. 2020. Microservices Pattern: Saga. Microservices.io. [Online] Available at: https://microservices.io/patterns/data/saga.html [Accessed 24 Feb 2020]
[26] Newam, S. (2014). Demystifying Conway's Law. ThoughtWorks, [Online] Available at: https://www.thoughtworks.com/insights/articles/demystifying-conways-law
[27] Dragoni N. et al. Microservices: Yesterday, Today, and Tomorrow. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. 2017, Springer, Cham pp 195-216
[28] (2015). The Future of Application Development and Delivery Is Now. nginx. [Online] Available at: https://www.nginx.com/resources/library/app-dev-survey/ [accessed 22 February 2020]
[29] L. Chen. Microservices: Architecting for Continuous Delivery and DevOps. In: 2018 IEEE International Conference on Software Architecture (ICSA). Seattle, WA, 2018, pp. 39-397. [Book] [accessed 17 february]
[30] Kharenko, A. (2015). Microservices Articles, Monolithic vs. Microservices Architecture. [Online]. Available at: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59 [accessed 17 february]
[31] Wolff, Eberhard. 2016. Microservices: flexible software architecture. Addison-Wesley Professional. [Book]
[32] Bolboaca, A .2019. Modular Monolith Or Microservices?. Moziac Works. [Online] Available at: https://mozaicworks.com/blog/modular-monolith-microservices/ [accessed 20 February 2020]
[33] Lumetta, J .(2018). These are the most effective microservice testing strategies, according to the experts. FreeCodeCamp [Online] Available at: https://www.freecodecamp.org/news/these-are-the-most-effective-microservice-testing-strategies-according-to-the-experts-6fb584f2edde/ [accessed 20 February 2020]
[34] Conklin, N. (2019). Medium, Modular Monoliths — A Gateway to Microservices. Medium. [Online] Available at: https://medium.com/design-and-tech-co/modular-monoliths-a-gateway-to-microservices-946f2cbdf382 [accessed 20 February 2020]
[35] Arshed,S.(2018). Monolithic vs SOA vs Microservices — How to Choose Your Application Architecture. Medium [Online] Available at: https://medium.com/@saad_66516/monolithic-vs-soa-vs-microservices-how-to-choose-your-application-architecture-1a33108d1469
[36] Singer, S. (2019). Miniservices, Microservices Rebalanced. Scanning Pages. [Online] Available at: https://scanningpages.wordpress.com/2019/03/04/miniservices-micorservices-rebalanced/
[37] The newstack, Miniservices: A Realistic Alternative to Microservices, 2018.[Online] Available at: https://thenewstack.io/miniservices-a-realistic-alternative-to-microservices/
[38] Eberhard Wolff, Self Contained Systems (SCS): Microservices Done Right, 2017. [Online] Available at: https://www.infoq.com/articles/scs-microservices-done-right/
[39] Assembling software from independent systems. Self-Contained Systems. [Online] Available at: http://scs-architecture.org/
[40] Annenko, O. (2016). Breaking Down a Monolithic Software: A Case for Microservices vs. Self-Contained. Elastic.io [Online] Available at: https://www.elastic.io/breaking-down-monolith-microservices-and-self-contained-systems/
[41] Chakrabarti, S. 2016. In the land of microservices, the network is the king. Medium, [Online] Available at: https://medium.com/lightspeed-venture-partners/in-the-land-of-microservices-the-network-is-the-king-maker-37de7ec4119a#.m3die0s7l [Accessed 19 February 2020]

[42] (2019). Now Powering Over 1 Million Merchants, Shopify Debuts Global Economic Impact Report. Shopify, Company News. [Online] Available at: https://news.shopify.com/now-powering-over-1-million-merchants-shopify-debuts-global-economic-impact-report-271485
[43] Shopify Unite Track. Deconstructing the Monolith - Kirsten Westeinde. (2019). Shopify Partners. [Online] Available at: https://www.youtube.com/watch?v=ISYKx8sa53g
[44] Fowler, M. 2007. Design Stamina Hypothesis. [Online]
Available at: https://www.martinfowler.com/bliki/DesignStaminaHypothesis.html
[45] Shatro, K. 2017.Upgrading Shopify to Rails 5. Shopify Engineering. [Online]
Available at: https://engineering.shopify.com/blogs/engineering/upgrading-shopify-to-rails-5-0
[46] Gjorgjievski, D. 2015. Understanding Scope in Ruby. SitePoint. [Online]
Available at: https://www.sitepoint.com/understanding-scope-in-ruby/
[47] Morgan, A. 2019. How Shopify Migrated to a Modular Monolith. InfoQ. [Online]
Available at: https://www.infoq.com/news/2019/07/shopify-modular-monolith/
[48] Westeinde, K. 2019. Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity. Shopify Engineering. [Online] Available at: https://engineering.shopify.com/blogs/engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity
[49] Ruby on Rails - Directory Structure. [Online] Available at: https://www.tutorialspoint.com/ruby-on-rails/rails-directory-structure.htm
[50] Netflix - Statistics & Facts, 2020, [Online], Available at:https://www.statista.com/topics/842/netflix/
[51]Smartbear Software, 2015. [Online] Available at:https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/
[52]Tom Macaulay, Ten years on: How Netflix completed a historic cloud migration with AWS,2018, [Online] Available at: https://www.computerworld.com/article/3427839/ten-years-on--how-netflix-completed-a-historic-cloud-migration-with-aws.html
[53]Brandon Butler, Netflix is (not really) all in on Amazon's cloud, 2016, [Online] Available at: https://www.networkworld.com/article/3037428/netflix-is-not-really-all-in-on-amazon-s-cloud.html [56] Mansoor Iqbal, Netflix Revenue and Usage Statistics, 2020,[Online]
Available at: https://www.businessofapps.com/data/netflix-statistics/
[54]2013 Netflix, Inc.[Online]
Available at: https://github.com/Netflix/Hystrix