A Comparison of Implementation issues of Monolithic vs Microservices Software Architectures

Nokuthula Manana
Department of Computer Science, University of Pretoria
Pretoria, South Africa
u12064115@tuks.co.za
siphokazi.manana@gmail.com

Vreda Pieterse
Department of Computer Science, University of Pretoria
Pretoria, South Africa
vpieterse@cs.up.ac.za

ABSTRACT

Microservices Architecture is a relatively new architecture upon which Software Engineers can build software. The whole premise of the architecture is to divide the different components of a system into small independently maintained modules in order to improve the testing and maintenance of a system. Microservices have been the buzz word in the industry for the past few years and some curious developers have been exploring this architecture as opposed to the more traditional monolithic architecture whereby the entire system is built and deployed at once, modules are inter-liked and upgrades to any part of the system would require the system to be reworked entirely. As the number of Software development companies in the world increase, it would be prudent to explore the advantages of microservices as an architecture and to what extent software engineering practices can benefit from its application. This article investigates the feasibility of using a microservices architecture as opposed to the more traditional monolithic architecture for software development.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Software performance**;

KEYWORDS

microservices architecture, monolithic architecture

ACM Reference format:
Nokuthula Manana and Vreda Pieterse. 2017. A Comparison of Implementation issues of Monolithic vs Microservices Software Architectures. In *Proceedings of COS700, Pretoria, South Africa, November 2017 (COS700 Journal Ariticle),* 12 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Software Architecture can be defined as the blueprint that guides *how* a system will be built and maintained. Although sometimes disregarded as a minor trade-off, Software Architecture can be the deciding factor in determining whether or not a software solution is successful. With the ever growing number of start-ups in IT industry, it would be prudent to accurately predict the sustainability of a software system by making an informed decision about the architecture on which the software will lie. This information is important because we would like to decrease the efforts in maintaining what could become a complex system as it evolves and requirements change in future. Being able to make these informed decisions early on in the development of a software system could benefit a company in a number of ways, one of which is to save the company large amounts in maintenance costs.

This research aims to dissect the semantics and implications of monolithic and microservice architectures by comparing and contrasting differences in metrics of the systems in the different architectures. An experiment is conducted in which two systems, *Bellisimo* and *SqueakyClean* are implemented.

SqueakyClean is assumed to be better suited for a monolithic architecture by nature of the application and in the same breath *Bellisimo* is assumed to be better suited for a microservices architecture.

Both SqueakyClean and Bellisimo will be implemented on *both* architectures so as to minimize the number of outside factors that could affect the experiment. The architectural and component level metrics of all four systems will be calculated and analysed in order to evaluate whether or not a change in architecture has an effect on the software quality and possibly maintenance of that software.THe information of this experiment is gathered in the hopes that a conclusion can be made about which architecture would be more suitable than the other for a particular application.

As we delve into this paper we will look at some formal definitions of monolithic applications and microservice applications. We will also explore the challenges developers face with these architectures, what has been proposed as solutions to these challenges and how this relates to the experiment conducted.

We will also dissect the experiment as a whole by exploring the methodology used, describing the specifications of the systems as well as contrasting the architectures.

Finally, we will take a qualitative approach and describe the implementation issues experienced while developing the systems. These will be my personal issues as well as the issues of the students who worked on one of the four systems for their COS731 module assignment.

2 BACKGROUND

Software Architecture can be defined as the blueprint that guides *how* a system will be built and maintained. Although sometimes disregarded as a minor trade-off, Sofware Architecture can be the

COS700 Journal Ariticle, November 2017, Pretoria, South Africa
© 2017 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06...$0.00
https://doi.org/10.475/123_4

deciding factor in determining whether or not a software solution is successful. With the ever growing number of start-ups in IT industry, it would be prudent to accurately predict the sustainability of a software system by making an informed decision about the architecture on which the software will lie. This information is important because we would like to decrease the efforts in maintaining what could become a complex system as it evolves and requirements change in future. Being able to make these informed decisions early on in the development of a software system could benefit a company in a number of ways, one of which is to save the company large amounts in maintenance costs.

In this section, we dissect the semantics and implications of monolithic and microservices architectures by comparing and contrasting their characteristics as described by other literary papers. We will also look at the different scenarios in which it is deemed that one architecture would be more suitable than the other.

2.1 Why Monoliths?

A Monolithic application is defined as a single-tiered software application in which the user interface, code logic as well as data access code are all combined into one program from a single platform. This unit is deployed as a whole and changes to code are infrequent and require redeployment of the entire system to ensure that things work. Monoliths are by definition self-contained and for the most part do not require interaction with a separate entity in order to function or serve its purpose. Monolithic applications are regarded as suitable for applications that are small and are not predicted to grow in size and complexity over the years.

Some of the limitiations that software developers encounter with monolithic applications are:

- **Stringent Testing Process:** If not developed under a Test-Driven Development (TDD) Framework, testing a monolithic application can become a daunting task. A few of the problem you could face are firstly debugging. The way in which the program has been coded could slow down your debugging process. A typical example would be to add breakpoints in a number of linked functions just to determine where in the code the application breaks. This is a tedious drawback that can be mitigated by enforcing software quality, separation of concerns in particular.
  Secondly, creating unit tests for the application after the logic has been coded and is functional could be difficult because you may find yourself creating more integration tests than unit tests because components may be so interlinked it would be impossible to test a method in isolation.
  Finally, you may find yourself doing more manual testing just ensure the software does what its supposed to do rather than doing any code quality testing because of business requirements or time constraints. In the long run this could be detrimental if the scope of the application increases and the quality of code has not been measured / has been measured poorly and there are no structures in place in order to consistently maintain code quality.
- **Tight Coupling:** Making a change to any aspect of the system could have adverse effects on another component of that system because components are so heavily dependant on one another.
- **Deployment:** In most cases whenever new features are added to the system or upgrades on the software tools being used are performed there will most likely be downtime experienced for the application. This could become a bottleneck for complex system software needing frequent system upgrades. Also, there usually has to be a stringent testing process before such an operation takes place in order to identify any adverse effects on the application by performing such operations.
- **Difficulty introducing new features:** Monolithic applications are known to be tightly coupled and many components are dependant on others in the software. Introducing a new feature to the application could become a difficult task as you need to keep track of what needs to change in all the different components of the application to accommodate the new feature. Also, a new feauture increases the complexity of the application as a whole and new bugs could be introduced into the application as a consequence.
- **Difficult to Maintain:** The problems that software developers typically face with monoliths is that they are difficult to deploy, upgrade and maintain. **citation** This is because changes in one part of the system could adversely affect another part of the system and thus deployments of major changes could take up to months to roll out into production. The typical occurence in industry is that new requirements are added to the system which require additions of new features and possibly alterations of existing ones. This increases the complexity of the system as a whole and the fast-paced change in the technology industry is one that is proving to be difficult for monolithic systems to keep up with.
- **Lack of documentation:** Another major issue with monoliths in industry today is that there is a lack of consistent documentation of the state of the system at any given moment. **citation** Any new developer brought onto the project will have to manually discover how the system works and this is a time consuming feat that could be costly to the company should they have tight deadlines to adhere to; which is often the case when additional human resources are brought onto a project.

Some of the advantages that monolithic applications have over microservices applications are:

- The advantage of Monoliths over Microservices is that there is no added overhead of interprocess communication or added machinery needed for the system to work. This could be seen as advantageous in systems that are low in complexity and properly managed, e.g. Good documentation, following coding standards and having automated tests to control the quality of the system.

2.2 Why Microservices?

The Software Development Life Cycle (SDLC) is a paradigm highly used in industry and traditionally is divided into four phases namely, *Requirements Gathering, Software Design, Implementation and Testing*. The ever growing and changing industry of IT is one that we cannot tame. The rate at which change in the market, technology and even trends requires quick responses from Tech Companies in order to keep up with demand. This ability to change at any given moment is crucial and is increasingly becoming a huge factor to take into consideration when embarking on a software development project. Unfortunately, these changes aren't predictable by the sofware architect at the beginning of a project, so how does one anticipate them and prepare accordingly should they occur? The proposed solution to this is Microservices.

Lewis **citation** has defined microservices as: *an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

This brings about a number of advantages to the maintainability of the system and directly mitigates all the problems stated previously about monolithic systems, namely *tight coupling, testing issues* and *deployment issues.*

A microservices architecture is synonymous with a distributed architecture. In a distributed architecture, subsystems are often physically separated and exchange resources through standard interfaces. They are used to increase system flexibility and responsiveness to change.The key factor to keep in mind when thinking about microservices is that these services are provided and maintained in isolation to one another. **citation** This improves the software correctness of a system and makes it easily testable and refactorable.

The following describes the benefits that an organization will have from implementing a microservices architecture **Morkel Theneussen**

- **Technology Heterogenity:**
- **Resilience:**
- **Scaling:**
- **Ease of deployment:**
- **Organizational Alignment:**
- **Composability:**
- **Optimizing for replaceability:**
- **Release Frequency and Agility:** Microservices architectures allow for frequent releases and quick response to software changes. Companies such as Amazon which use microservices can make over 1000 changes per day and these can be reflected on their production environment due to the ability to release frequently that microservices provides to developers. Monolithic systems, on the other hand, could take several months to reflect software changes. With large development teams, this can result in merging conflicts, testing issues and conflicts and this can result in a huge, unreliable process. Microservices can solve this problem by skipping the merging process so that individual teams can test their small packages or services and easily deploy them. This is a simple way to change code without negatively altering the rest of the system and the decoupling nature of microservices allows for this to occur.

On the other side of the coin, Microservices do have a few tradeoffs to take into consideration. According to A. Singleton **citation** there are substantial drawbacks of a Microservices architecture that should be taken into consideration, namely:

- **System Size and complexity:** Microservices Architecture is generally seen as suitable for bigger, more complex systems rather than smaller, simpler systems. According to Singleton, it would be at one's detriment to implement a simple system with a microservices architecture because there will be an added unnecessary overhead of communicating with different services, not to mention routing to services, deploying services and the possible extra machinery needed. One could save time and money by building a monolithic system if the system is both simple and small.
- **Service Monitoring and logging:** Monitoring the different services can become a difficult task for the developer. Each of these different services are most likely going to use different technologies or languages; reside on different machines or containers (thus having seperate logging mechanisms) and have its own version control. This as a consequence makes the system as a whole fragmented and for the purpose of monitoring the system and viewing its state in its entirety a need may arise for centralized monitoring as well as using tools to centralize the different logs emitted by the different services.
- **Determining Root Cause:** Finding the root cause of a problem within a microservices architecture could become problematic if that error happens to span over multiple services.
- **Version Management and Circular Dependencies between services:** In some cases you may find that multiple services share a dependency on a service that needs an update, you may have to update the API's of the other services. This can bring about questions like which service should be updated first or how does one make a safe transition when performing the update. The more services you have the more release cycles for each that have to be managed and this adds to this complexity of the system. Reproducing a problem will prove to be very difficult when it is gone in one version, and returns in a newer one.
- **Maintainability:** Deploying with multiple small components can be a cumbersome task especially for smaller systems that are not as complex as what microservices assume.
- **Skillset:** As this is a relatively new paradigm, the required skillset to work with microservices technology may be limited and thus could be costly to the company.

One consideration to note is that determing which architecture to use is a process that involves a number of factors. The first and most important factor is to determine the scope of the application. Is it a complex system that has many different components acting together or is it a simple application or website. Does the company expect the software to grow or evolve in any way and how frequently do they expect changes to occur within the application.

Other key factors to note when deciding on an architecture are **Cite A. Singleton here**

- **Build vs Buy:** Software under a microservices architecture is more integratable with web services offered by cloud systems. This allows a company to delegate build and operational costs to an external vendor providing SaaS (Software as a Service) at an fee. Going the microservices route could set an easier platform for a company to buy services than with a monolithic system considering that cloud platforms are more inclined towards that type of architecture.
- **One Product or Many Products:** Microservices are useful when supporting multiple products and services. This is because microservices allow one to reuse services in more than just one product, unlike monolithic software systems which has custom code that is tightly coupled to other services in one application.

As we delve further into the intricasies of Microservices and Monoliths, the aim is to gain insight in order to make informed architectural decisions about Software Development.

Taking all this information into consideration, we can clearly see that a Microservices Architecture should not be seen as a silver bullet that will solve all problems that developers have with monolithic architectures. Instead, the developer should realise that based on the business model of the application that must be built, a number of factors such as the ones mentioned above should be taken into consideration when deciding on an architecture to implement.

3 RESEARCH DESIGN

3.1 Problem statement

Architectural design decisions are often made based on the current trends and without proper insight about the consequences of such a decision. Theoretical knowledge and understanding of the rational behind the development of known architectural design patterns may help developers to make better decisions.

While anecdotal evidence for and against the use of these architectures are abundant, very little empirical results can be found to support such claims.

This paper compares Monolithic Architectures with Microservices Architectures in order to provide empirical observations to strengthen insight and understanding into the different aspects of the software architectures under investigation. These insights have the potential to support developers when they have to make architectural decisions related to the design of their software systems.

3.2 Criteria for comparison

The following criteria set will be applicable to the systems that will be created.

3.2.1 Task Completion. For every phase of the SDLC the amount of time it takes to complete a particular task should be measured. This will be contrasted with how long it takes to implement those same features on the different architecture.

3.2.2 Software Quality. This criteria will be measured after the implementation phase of the research has been conducted. The following list describes the metrics that will be used to assess Software Quality of each system.

Architectural Design Metrics

These metrics focus on the characteristics of the program architecture, emphasizing the architectural structure and effectiveness of the modules/components therein. Three software design complexity measures are identified, namely:

- Structural Complexity:

  $$S(i) = f_{out}^2(i) \tag{1}$$

  where i is the module being measured and f_{out} is the fan-out value of that module.

  Fan-out is defined as the number of modules immediately subordinate to the module *i*, that is, the number of modules that are directly invoked by module *i*.
- Data Complexity:

  $$D(i) = \frac{v(i)}{f_{out}(i)+1} \tag{2}$$

  where $v(i)$ is the number of input and output variables that are passed to and from the module *i*.
- System Complexity:

  $$C(i) = S(i) + D(i) \tag{3}$$

  As each of the complexity values increases, so does the overall architectural complexity of the system. This leads to a greater likelihood that integration and testing efforts will also increase.

Component-level Design Metrics

These metrics focus on internal characteristics of a software component and include measures of the "three Cs" namely module complexity, coupling and cohesion.

COMPLEXITY

In this research cyclomatic complexity is what will be measured to get the indication of the complexity of the system. According to McCabe and Watson [2], some of the important uses for complexity metrics are:

- To predict critical information about the reliability and maintainability of software systems
- To provide feedback during the software project to help control the design activity
- During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability.

The three methods which we will use to calculate and verify the calculations of the cyclomatic complexity are:

- Creating a flow graph and calculating the number of regions in that flow graph. Regions correspond to the cyclomatic complexity.
- Cyclomatic complexity *V(G)*, for a flow graph, *G*, is defined as

  $$V(G) = E - N + 2 \tag{4}$$

  where E is the number of flow graph edges, and N is the number of flow graph nodes.

- Cyclomatic complexity, $V(G)$, for a flow graph G, is also defined as
$$V(G) = P + 1 \tag{5}$$
where P is the number of predicate nodes contained in the flow graph G. A predictae node is also known as a diamond/conditional node.

COUPLING

Coupling is a software metric that determines the relatedness of modules to one another. As a software development rule, high coupling is considered detrimental to the maintainability and testability of software. In general, the aim is to get the coupling between modules to be as low as possible because this lowers the possibility that a change within a module will affect the module that is seen to have a high coupling factor to it[1].

Coupling will be measured according to Dharma's method in their published paper: *Quantitative measures of Cohesion and Coupling in Software Development.*

Let
C = total module coupling

For data and control flow coupling:
$i1$ = in data parameters
$i2$ = in control parameters
$u1$ = out data parameters
$u2$ = out control parameters

For global coupling:
g1 = number of global variables used as data
g2 = number of global variables used as control

For environmental coupling:
w = number of modules called
r = number of modules calling the module under consideration.

Now we let
$$m = i1 + (q_6 \times i2) + u1 + (q_7 \times u2) + g1 + (q_8 \times g2) + w + r \tag{6}$$
where q_6, q_7, q_8 are constants and as a first heuristic are assumed to be 2 in accordance with Dharma's calculations [**?**]. In their paper, the premise is that control variables result in twice as tight a coupling as an equal number of data variables. Thus the constants applied to the calculations in this paper will also be 2.

The minimum value of r is 1 because every module executed must be called by some other program.

Now, C is inversely proportional to m ($C \propto \frac{1}{m}$)

therefore $C = \frac{k}{m}$ where k is a constant. Assuming $k = 1$ we have:
$$C = \frac{1}{m} \tag{7}$$
Before we delve into the calculations for coupling lets iron out some definitions of key terms first.

- **Data parameters:** These are parameters that hold valuable data within that method. This data would consist of output variables, global variables and internal data variables that work solely in that function.
- **Control parameters:** Variables that determine operational execution such as when used in a loop or if-then type of statement A variable used as a diagnostic or error recovery variable is also classified as a control variable.
- **Data coupling:** The relatedness of data variables to one another
- **Control flow coupling:** The relatedness of control variables to one another.

COHESION

Cohesion within a module refers to the strength of the relationship between pieces of functionality within a given module; it refers to the degree to which the elements inside a module belong together. It is known that the more cohesive a module is, the easier it is to maintain and test that module. Therefore high cohesiveness is a general aim within software engineering.

In this research, slice-based metrics will be used to measure cohesion, in the same way that Binkley and Meyers calculated their cohesion for their research paper [3]. A slice can be defined as: *A backward walkthrough a module that looks for data values that affect the state of the module when the walk begain. This can be in the form of statement variables or conditional variables withing the module. [2]*

The slice-based metric is comprised of five smaller measurements that relate to each data slice within that module, these measurements are

- **Tightness:**
$$Tightness(M) = \frac{|SL_{int}|}{length(M)} \tag{8}$$
Measures the number of statements in every slice
- **MinCoverage:**
$$MinCoverage(M) = \frac{1}{length(M)} min_i |SL_i| \tag{9}$$
The ratio of the smallest slice in a module to the module's length
- **Coverage:**
$$Coverage(M) = \frac{1}{|V_O|} \sum_{i=1}^{V_O} \frac{|SL_i|}{length(M)} \tag{10}$$
Compares the length of slices to the length of the entire module
- **MaxCoverage:**
$$MaxCoverage(M) = \frac{1}{length(M)} max_i |SL_i| \tag{11}$$
The ratio of the largest slice in a module to the module's length
- **Overlap:**
$$Overlap(M) = \frac{1}{|V_O|} \sum_{i=1}^{V_O} \frac{|SL_{int}|}{|SL_i|} \tag{12}$$
A measure of how many statements in a slice are found only in that slice.

Note that V_M is the set of variables used by module M and V_O is the subset of V_M that are output variables. Output variables can be considered those variables that are a functions return value or

those that are globally modified by that function.
SL_i is the slice obtained for $v_i \in V_O$ and SL_{int} is the intersection of SL_i over all $v_i \in V_O$

These measurements used together can describe the cohesion of a module.

3.3 Experimental systems

3.3.1 Description: In order to conduct this experiment fairly, four simple web applications are be created. The first two web applications are chosen to be well suited for a monolithic architecture. The difference between these systems is the architecture on which they are designed, i.e. one on a monolithic architecture and the other on a microservices architecture.

The other two web applications are chosen to be well suited for a microservices architecture. These are also be designed differently. One is designed on a monolithic architecture while the other is on a microservices architecture.

In both instances the logic of the applications will remain the same.

3.3.2 Monolithic-suited Application - SqueakyClean: Accompany is a simple company website that provides information about the company, its vision and its trade. This website is intended to be a static, informative site. The core of the system will be a single page application that dynamically loads different content describing the company based on which tab is selected. The website will allow any user to be able to send a query to the company to find out more information.

Design Specifications: The system should be designed in such a way that the mailing service can easily be plugged in and out of the system i.e. it should be loosely coupled from the rest of the website. The following are the design specifications for the website:

- Any updates to the website will be handled by the administrator.
- The Anonymous User interface will allow people to browse the entire website
- The mail service that will enable users to enter their contact details should they want to be contacted by the company. The mail service should send an email or SMS notification to the relevant people within the company to handle queries.

Design Technologies: The following technologies will be used to implement the system:

- Html 5 (Html and Bootstrap CSS)
- Angular2
- Spring Boot
- Spring MVC
- Apache Maven
- Git (Github)
- PostgreSQL Database
- Spring Cloud Zuul

3.3.3 Microservices-suited Application - Bellisimo: Bellisimo is a fictional company aimed at providing an online platform for customers to browse the clothing as well as food catalogues. Information about specials and promotions will be published on this platform. Bellisimo is assumed to have two main focus areas within the business namely, to be a clothing store as well as a supermarket.

The core of the system will be a catalogue of items and their prices. Since Bellisimo is involved in clothing and food, the catalogue will have to ensure that these regions are well maintained. Sales and specials in each region will have to be accounted for and managed. The scope of the system is to ensure that the latest information is being provided about items and their prices.

Design Specifications: The system should be designed in such a way that the clothing module is mutually exclusive to the food module.That means that the clothing module can be deployed and maintained separately from that of the food module while still maintaing that this information is delivered by one website. The following design specifications are what the system should adhere to:

- There should be at least two interfaces. An admin interface to maintain the catalogue and an anonymous user interface that will allow people to interactively view the catalogue.
- The prices of each item in the database should be displayed along with the image of the item.
- The administrator should be able to add, remove and update any items in the catalogue list.
- The administrator should be able to add specials to the module. Specials can be applied to singular items or groups of items depending on a special group configuration.
- Users should be able to browse, search and filter the catalogue.

Design Technologies: The following technologies will be used to implement the system:

- Html 5 (Html and Bootstrap CSS)
- Angular2
- Spring Boot
- PostgresSQL
- Apache Maven
- Git (Github)
- Spring Cloud Zuul

4 DATA ANALYSIS

4.1 Architectural Metrics

Table 1 show the results of the architectural metrics measured for the four systems.

4.1.1 SqueakyClean Analysis. Looking at the results of the calculations it would seem that the systems overall complexity has increased in the microservices architecture when compared to the monolithic implementation. This result is in line with our predictions because SqeakyClean is regarded as a system that is better suited for a monolithic architecture.

4.1.2 Bellisimo Analysis. When comparing the Microservices implementation of Bellisimo to that of the Monolithic Implementation, these observations are noted and analysed:

- The Structural Complexity is lower in the microservices architecture when compared to the monolithic for the complex ItemService module.

System	Module (*i*)	Metric	Monolithic	Microservices
SqeakyClean	ClientService	Structural Complexity	4	16
	MailService		0	4
	ClientService	Data Complexity	2/3	2/5
	MailService		2	2/3
	ClientService	System Complexity	4.67	16.4
	MailService		2	2.67
Bellisimo	ItemService	Structural Complexity	49	36
	SpecialService		25	25
	ItemModule	Data Complexity	1.5	1.71
	SpecialModule		1.33	1.33
	ItemModule	System Complexity	50.5	37.71
	SpecialModule		26.33	26.33

Table 1: Architectural Metrics for the four systems

- The simple SpecialService Module had no change in its structural complexity. This is because the code base for the special service did not change at all in both architectures.
- The Data Complexity in the Microservices implementation increased for the ItemService when compared to the Monolithic implementation. This is because the added communication layer to retrieve an object from the SpecialService required an addition to the number of data objects entering the module.
- The data complexity of the SpecialService did not change across the architectures because the code base of the Service did not change.
- Thus the system complexity of Bellisimo is Greater when implemented in a monolithic architecture as compared to the microservices architecture.
- This leads to the conclusion that a Microservices architecture is better suited for an application such as Bellisimo.

4.2 Component-level Metrics

Component software metrics measure the attributes of a software component. A component can be regarded as being as small as a procedure or as large as a module. In this research we are going to measure the component level metrices in terms of the different modules that can be identified in the software. E.g. ClientService

Component level metrices will be used in this research to possibly make a correlation between architecture change and change in complexity of the software itself. This may/may not give an indication of whether microservices architecture has an impact on the overall software quality.

The component metrics that are going to be measured are *cohesion, coupling* and *cyclomatic complexity.*

4.2.1 ***Cohesion****.* For the purposes of this research, the cohesion metrics will be calculated for the relevant method being called in the **Service** class of that component. The controller and exception handling classes will be considered out of scope for the purpose of this research. The reason for undertaking in this decision is that the majority of the business logic of the applications will reside in the *Service* classes. The other classes are seen as too static to be evaluated for these metrics.

Table 2 show the results of the cohesion metrics measured for SqueakyClean.

The ClientService had the same metrics for cohesion in both architectures. Thus indicating that a change in architecture has little to no effect on the cohesion of the module. This, however, must be taken with a pinch of salt as the level of granularity of measurement may be a contributing factor in that result.

Table 3 show the results of the cohesion metrics measured for Bellisimo in the ItemService for the Monolithic implementation.

Table 4 show the results of the cohesion metrics measured for Bellisimo in the SpecialService for **both** implementations.

With the introduction of the *restTemplate* object that will communicate between the services in the microservices architecture, the cohesion metrics of Bellisimo differ slightly in the different

System	Module	Method	Metrics	Monolithic	Microservices
SqeakyClean	ClientService	storeClientInformation	Tightness	17	17
			MinCoverage	33	33
			Coverage	50	50
			MaxCoverage	67	67
			Overlap	42	42
	MailService	sendMail	Tightness	7	6
			MinCoverage	14.2	13
			Coverage	54	53
			MaxCoverage	93	94
			Overlap	29	28

Table 2: Cohesion metrics for Squeaky Clean in both Architectures

architectures. Understandably, the microservices architecture is less cohesive than the monolithic architecture.

4.2.2 ***Coupling***. Table 5 shows the results of the coupling within the 4 systems created.

Looking at the SqeakyClean System, it seems that the coupling in the microservices implementation is lower than that of the monolithic implementation of the same system.

SqeakyClean Analysis
Based on the Coupling calculations computed for SqueakyClean, the following was observed:

- The coupling metrics for the microservices architecture was less than that of the monolithic architecture.
- This leads to the conclusion that a change in architecture may have affected the coupling of the system. However it is prudent to note that this could only be applicable at this level of granularity. If a module were to be regarded as more than just the service class, it is possible that the coupling metrics could differ from those observed in this instance.
- Based on that, I can then conclude that change in architecture has an effect on coupling.

Bellisimo Analysis
Based on the Coupling calculations computed for Bellisimo, the following was observed:

- The coupling metrics stayed exactly the same for both implementations.
- This leads to the conclusion that a change in architecture may not significantly affect the coupling of the system. However it is prudent to note that this could only be applicable at this level of granularity. If a module were to be regarded as more than just the service class, it is possible that the coupling metrics could differ from those observed in this instance.
- Based on that, I can then conclude that change in architecture has no effect on coupling.

4.2.3 ***Cyclomatic Complexity***. Table 6 depicts the results of the cyclomatic complexity calculation of the different systems.

SqeakyClean Analysis
Looking at the complexity metrics of the SqeakyClean system, it seems that the change in architecture did not affect a change in the system.

Bellisimo Analysis
The component level complexity of Bellisimo is the same on both architectures. Similar to the couping metrics of the systems this could be because the level of granularity of the measurements were not significant enough to see changes based on the different architecture, or it could be that architectural changes have no effect on the component-level metrics. Or better yet, there is no correlation between the difference in architecture and difference in the component level metrics.

4.3 Qualitative Analysis

A qualitative analysis of the research is conducted in order to guage the amount of effort that would be typically required of a developer to implement a system in each architecture. In this section I summarize the experiences with implementing the systems under the specific architecture as well as compare the manual difficulties faced during the development process.

4.3.1 SqueakyClean Monolithic. Since sqeaky clean was a relatively simple implementation to develop, the creation of the pages were very simple and easy. The challenging part of this implementation was the use of the new technologies. All of the technologies chosen for the research I have less than 1 year experience with them so when solving bugs in the system, there was plenty of time taken by researching technology specific solutions.

4.3.2 SqueakyClean Microservices. Since the logic of the system was already implemented, migrating the system to the microservices architecture was fairly simple however I had to create an extra controller class for the mailservice in order to access it from the client module. This addition was not necessary with the monolithic architecture because the mail service was injected into the client service for use. No api calls were expected to be made to the mail service thus no controller was created in the monolithic implementation. The most complex part of this implementation was setting up the integration service and ensuring that the microservices were

System	Module	Method	Metrics	Monolithic	Microservices
Bellisimo Monolithic	ItemService	addItem	Tightness	11	11
			MinCoverage	22	22
			Coverage	50	50
			MaxCoverage	78	77
			Overlap	32	32
		updateItem	Tightness	9	9
			MinCoverage	18	18
			Coverage	50	39
			MaxCoverage	81	81
			Overlap	31	37
		getAllFoodItems	Tightness	0	1
			MinCoverage	1	1
			Coverage	1	1
			MaxCoverage	1	1
			Overlap	0	0
		getAllClothingItems	Tightness	0	1
			MinCoverage	1	1
			Coverage	1	1
			MaxCoverage	1	1
			Overlap	0	0
		getItem	Tightness	0	29
			MinCoverage	86	29
			Coverage	86	64
			MaxCoverage	86	100
			Overlap	0	64
		deleteItem	Tightness	0	0
			MinCoverage	1	0
			Coverage	1	0
			MaxCoverage	1	0
			Overlap	1	0
		addSpecialToItem	Tightness	18	18
			MinCoverage	36	27
			Coverage	55	48
			MaxCoverage	73	81
			Overlap	38	32

Table 3: Cohesion metrics for Bellisimo - ItemService

all up. One major factor of microservices that I noticed in this experiment is that service monitoring is important and necessary. If for some reason a service were to failover, there should be measures in place set to notify the developer of the failover, also there should be an automated process that should fire to handle failovers.

4.3.3 Bellisimo Monolithic. Bellisimo was a more difficult implementation to complete as compared to SqueakyClean. Many features had to be catered such as the CRUD of items and specials, adding specials to items and recalculating an items value based on the special condition. The most challenging part of this implementation would be coding the front end application, user-friendliness and responsiveness were difficult to implement. I believe that the implementation of the backend application came easier due to the added experience with the technologies over the past few months.

Far less time was spent on research with the backend technologies when compared to SqueakyClean, but at the same time, far more time was spent implementing the application as compared to

System	Module	Method	Metrics	Monolithic	Microservices
Bellisimo	SpecialService	add	Tightness	13	13
			MinCoverage	25	25
			Coverage	50	50
			MaxCoverage	75	75
			Overlap	33	33
		update	Tightness	25	25
			MinCoverage	50	50
			Coverage	50	50
			MaxCoverage	50	50
			Overlap	50	50
		getAll	Tightness	0	0
			MinCoverage	1	1
			Coverage	1	1
			MaxCoverage	1	1
			Overlap	0	0
		get	Tightness	0	0
			MinCoverage	1	1
			Coverage	1	1
			MaxCoverage	1	1
			Overlap	0	0
		delete	Tightness	0	0
			MinCoverage	1	1
			Coverage	1	1
			MaxCoverage	1	1
			Overlap	0	0

Table 4: Cohesion metrics for Bellisimo - SpecialService Module

System	Module	Monolithic	Microservices
SqeakyClean	ClientModule	10	8
	MailModule	33	17
Bellisimo	ItemService	4	4
	SpecialService	6.3	6.3

Table 5: Coupling metrics for the four systems.

SqueakyClean due to the fact that the application had more functionality then SqueakyClean.

4.3.4 Bellisimo Microservices. The microservices implementation of bellisimo was simple to complete because most of the functionality remained the same. The only major difference was that the communication between the two systems had to be facilitated by the restTemplate. The most challenging aspect of this system was getting the services to communicate with one another. When looking at the time it took to configure the API Gateway, I would say the biggest drawback with Microservices architecture would be to orchestrate and maintain the integration server. Without some automated process to startup services and handle server failover, the job can become very verbose and tedious.

Table 7 is a summarized log of all the hours spent on the experiment. The time has been subdivided into research, implementation and bug fixing hours. Refer to *Appendix A* for a detailed breakdown of the tasks performed during these hours.

From the table we can note the following:

System	Module	Method	Monolithic	Microservices
SqeakyClean	ClientModule	storeClientInformation	2	2
	MailModule	sendMail	2	2
Bellisimo	ItemService	addItem	4	4
		updateItem	8	8
		getAllFoodItems	1	1
		getAllClothingItems	1	1
		getItem	2	2
		deleteItem	1	1
		addSpecialToItem	3	3
	SpecialService	add	2	2
		update	3	3
		getAll	1	1
		get	1	1
		delete	1	1

Table 6: Complexity metrics for the four systems.

	SqueakyClean Monolithic	SqueakyClean Microservices	Bellisimo Monolithic	Bellisimo Microservices	Total
Research and previous coding experience	78.75	16.5	78.5	15	**189**
Development	15.5	8	35	29.5	**88**
Bug Fixes	4	5	1.5	4	**14.5**
Total	**98.25**	**29.5**	**115.25**	**48.5**	**291.5**

Table 7: Hours spent on the experiment.

- More time was spent on Bellisimo than SqueakyClean. **Understandably because Bellisimo was a more complex system compared to SqueakyClean**
- The monolithic architectures took longer to complete than the microservices ones. This is because on each occasion, the monolithic architecture was tackled first. The implementation of the microservices required the duplication of most of the code logic. The aspect that took most time in the microservices imlementations were the configurations so that the services can communicate with each other.
- Most time recorded was on the previous coding experience. I believe that previous knowledge of the technologies and language is an important factor to note. In order for the complete implementation of approx. *100 hours* of code, atleast *150 hours* of background knowledge was needed. In my opinion, this also contributed to how low the big fixing hours are. The familiarity with a platform and language reduces the number of bugs that will be created during development.

5 CONCLUSION

We write this at the end.

6 FUTURE WORK

This section describes all the processes that could have been better implemented in this experiment and how they will be tackled in the future. There is also mention of enhancements that could yield better results for the aim of the experiment.

6.1 System's reaction to change

In order to measure the most agile architecture, a change factor needs to be introduced into the system. The way in which this system can adapt and accomodate the change factor will be recorded and analysed. Also, the quality of the software will be reevaluated in order to see if the change factor had any effect on the overall quality of the system. This will be useful in determining which architecture to use in industry where changes to the code base happen on a regular basis.

6.2 Improvement on the technologies used

There were a number of drawbacks experienced from this experiment. To name a few:

- There was no monitoring of services in place to note when a service failover occured. I had to manually check that each service was still up and running successfully.

- There is no automated process configured for setting up the services on the API Gateway, possible investigation in using *Travis* to automate this process is neccessary.
- Investigate the feasibility of using docker containers to demonstrate the full advantage of using microservices.

As this research grows, more efforts will be placed in using tools that are commonly known to aid the process of developing in a microservices architecture.

6.3 Extend experiment to students

A study can be introduced whereby students are given the opportunity to replicate the experiment by chosing one system to implement in one architecture and record their process just as was done in this experiment. The different results of all students can then be gathered and interpreted accordingly. This could aid in getting a more realistic experience of what was gathered in this experiment. A qualitative analysis may also be required of the students in order to gain insight into how the developers view the ease of using each architecture.

6.4 Measure Metrics on different levels of granularity

Since the component-level metrics did not yield usable results, in future experiments there could be multiple levels of granularity to measure the metrics on. The aim is to find a correlation in the component-level software metrics to that of the arcitectural metrics, should one exist.

REFERENCES

[1] H. Dhama. 1995. Quantitative Models of Cohesion and Coupling in Software. (1995).
[2] R.S. Pressman. 2005. *Software Engineering : A Practitioner's Approach.* McGraw Hill.
[3] D. Binkley T.M. Meyers. 2004. Slice-Based Cohesion Metrics and Software Intervention. (2004).