Bachelor's thesis

Bachelor's Programme in Computer Science

Comparing Monolithic- and Microservice Architecture through the lens of Quality Attributes

Alexander Engelhardt

December 18, 2023

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Bachelor's Programme in Computer Science

Tekijä — Författare — Author

Alexander Engelhardt

Työn nimi — Arbetets titel — Title

Comparing Monolithic- and Microservice Architecture through the lens of Quality Attributes

Ohjaajat — Handledare — Supervisors

Docent Jeremias Berg

Työn laji — Arbetets art — Level Bachelor's thesis	Aika — Datum — Month and year December 18, 2023	Sivumäärä — Sidoantal — Number of pages 28 pages, 8 appendix pages

Tiivistelmä — Referat — Abstract

Choosing the right software architecture is foundational when developing software-intensive systems. Two prevalent architectures that are both widely used are *Monolithic-* and *Microservice architecture*. Monolithic systems have in common that they all function on a single code base. In Microservice architecture we see an opposite philosophy, where one strives to partition a program into minimal independent *microservices* that are combined to work as a single system. To this end, Monolithic- and Microservice architecture make out the opposite extremes when considering how granular a software is, in that granularity describes to what degree a software is split into individual smaller programs. Constituting two extremes from an architectural design perspective, as well as both architectures being widely popular poses the question of how they differ. Additionally, several studies have identified the Monolithic- vs. Microservice discussion to be subject to a lot of nuance, including contradictory evidence on how they compare. Through comparing a selection of important *quality attributes* this thesis aims to shed light on some of the most striking differences between Monolithic- and Microservice architecture. By using quality attributes that highlight different aspects of software and software development a wide set of *stakeholder perspectives* are covered. The result is a big picture view which can help in choosing the right architecture for a software.

The core finding of this thesis is that Monolithic- and Microservice architecture express different strengths and weaknesses, each having unique use-cases. Most of the differences originate from the difference in granularity found in each architecture's core structures. Monolithic architecture is simpler as long as the software is not too large. When a software is larger, Microservice architecture gives added control of the software which can be a huge benefit.

ACM Computing Classification System (CCS)
Computer systems organization → Architectures → Distributed architectures
Software and its engineering → Software organization and properties → Software system structures → Software architectures

Avainsanat — Nyckelord — Keywords

Software Architecture, Microservice, Monolith, Quality Attributes, Granularity

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

The architecture of a software makes up its basic structure and has major implications on what properties can be achieved with that software (Bass et al., 2003). Thus it is important to choose the correct architecture. Monolithic- and Microservice- are two, widely used software architectures that differ fundamentally in their structure (Gos and Zabierowski, 2020). A Monolith is composed of a single code base (Richards and Ford, 2020). A Microservice based system consists of several smaller programs combined through *communication* (Dragoni et al., 2017).

In the scientific community there is a lot of dissonance on what the typical characteristics of Monolithic- and Microservice architecture are. For example Blinowski et al. (2022) have identified inconsistent evidence on how well the two architectures perform under different workloads. Al-Debagy and Martinek (2018) have made similar observations. Both studies imply that specific use-cases have great impact on what is a more suited architecture to use. This poses for an interesting dynamic to research. Additionally, studies often focus on only some aspect of the architectures, leading to a lack of the big picture of the different *use-cases* for Monolithic- and Microservice architecture. This is the niche that this thesis aims to partially fill.

To achieve a big picture view, an analysis is performed of how the fundamental structural differences between the two architectures affect different stakeholders, like end-users, developers, and businesses. More specifically, a comparison of a selection of *quality attributes* is made. Quality attributes are defined as *properties that can be used to indicate how well a system satisfies the needs of its stakeholders* (Bass et al., 2012). The selected quality attributes are *performance, scalability, deployability*, and *security* and have been chosen on the basis of contributing a wide set of different views relevant to different stakeholders. To retain the focus on use-cases, this thesis will not only consider theoretical capabilities but also look at practical challenges which might prevent realizing a theoretical potential.

The main finding of this thesis is that the fundamental difference in structure between Monolithic- and Microservice architecture forms a pattern of suitability for different use-cases. This pattern shows that Microservice architecture, when compared to Monolithic architecture, gives greater control of a system, but with added complexity. Additionally, the pattern of different use-cases, shows that the benefits of Microservice architecture are not realized as long as a system remains small. When a system grows larger the benefits become more apparent, but at the same time the added practical challenges associated with Microservice architecture might outweigh the benefits. The take-away from this finding is that when choosing between Monolithic- and Microservice architecture one needs to make a careful approximation of return on investment for using Microservice architecture, taking into consideration the real benefits and capability to handle the added complexity.

The layout of the thesis is the following: Central concepts important to the context are explained in Chapter 2. Chapter 3. first details how the chosen quality attributes highlight different aspects of software and software development and then compares how these quality attributes are expressed in Monolithic- and Microservice architecture. Chapter 4. presents and analyzes the results of the comparison and thus produces the main findings of this thesis. Chapter 5. concludes the thesis. Chapter 6. explains how this thesis has made use of artificial intelligence and large language models.

2 Context and Concepts

2.1 Software Architecture and Quality Attributes

As displayed by the long list of definitions for *software architecture* upheld by Carnegie-Mellon-University-Software-Engineering-Institute (2017), there is no single right definition we can use. According to *ISO/IEC/IEEE 24765:2017* (2017) software architecture is a software system's *fundamental concepts and properties in its elements, relationships, and in the principles of its design and evolution.* According to Solms (2012), software architecture is *the software infrastructure that an application needs to be able to specify, deploy, and execute user functionality.* Another definition is that software architecture is *a set of structures that are needed to reason about a system, consisting of software elements, relations among them, and the properties of both* (Clements et al., 2010).

In light of these definitions and the fact that there is no single *right* definition we can use, this thesis aims to synthesize a new definition based on the specific needs of this thesis. As the focus of this thesis is on comparing two software architectures, and how the choice of architecture affects key stakeholders the definition needs to highlight the relation between this choice and its implications. To formulate a definition suiting these needs, the profound role of *quality attributes*, as a driving force for choosing any software architecture needs to be explained in greater detail.

It can be argued that quality attributes, i.e. properties that can be used to indicate how well a system satisfies the needs of its stakeholders, lie at the heart of why there even are different software architectures. First, quality attributes are achieved by the basic structures designed into a software itself (Bass et al., 2012). Second, different quality attributes are often in conflict where improving on one attribute hurts another (Egyed and Grunbacher, 2004). They also tend to form complex inter-dependencies (Egyed and Grunbacher, 2004). Understanding the complex trade-offs and inter-dependencies between quality attributes and how these are tied to the core structure of a software is thus fundamental to understanding software as a whole. This is also reflected by Bass et al. (2012) who describe quality attributes as a key driving factor when deciding what software architecture to use. Based on the notion that the core structures of a software has major implications on what can be achieved with that software, the working definition for the concept of *software architecture* in this thesis is the following:

The architecture of a software is a skeleton-like structure which inherently allows for achieving certain quality attributes over others.

This definition aligns with the goals of this thesis, emphasizing that fundamental structural differences in architecture lead to variations in quality attributes, that in turn, reflect the needs of stakeholders. The close relationship between the *structure of a software*, *quality attributes* and *stakeholder satisfaction* also functions as motivation for using quality attributes as the core lens through which the comparison of Monolithic- and Microservice architecture is performed.

2.2 Granularity of Monoliths and Microservices

Monolithic architecture is another word that lacks a commonly accepted definition. Richards and Ford (2020) use the term monolithic to refer to a whole category architectures that are deployed as single unit. Deployment can be defined as the "phase of a project in which a system is put into operation and cut-over issues are resolved" (*ISO/IEC/IEEE 24765:2017* 2017). Other sources speak of it as its own architecture without mention of category (Gos and Zabierowski, 2020), (Al-Debagy and Martinek, 2018). They do however agree with Richards and Ford (2020) on the fact that Monolithic software consists of *one code base that is deployed as a single standalone program.* This is also a key aspect of how this thesis uses the term Monolithic architecture and the influence of deployable units on quality attributes is analyzed. In this thesis the term *Monolithic architecture* is used to refer to the theoretical concept, while the term *Monolith* is used to describe a software system which is based on a Monolithic architecture.

Microservice architecture is a way of combining minimal independent programs through communication, which together form a coherent system (Dragoni et al., 2017). For example, we could have a website which offers news, weather-forecasts and a discussion forum. In a *Monolith* all of these would be located in the same code base, but in a *Microservice based system*, we can split these aspects into separate code bases that are combined through communication to form a coherent program. The number of individual microservices in a Microservice-based system can vary from just a few to hundreds or even thousands of individual microservices (Gan and Delimitrou, 2018). The separate minimal programs of a Microservice based system are deployed independently of each other. A key concept within Microservice architecture is the possibility of using *heterogeneous technology*, which entails that the individual microservices can use completely different technologies (Rademacher et al., 2019). This is in contrast to Monolithic architecture where all parts of the system must use the same base technology. In this thesis, the term *Microservice architecture* refers to the theoretical concept, while *Microservice-based system* refers to a system that uses *Microservice architecture.* The term *individual microservice* refers to the individual, independent smaller programs that together form the unified software.

The granularity of software is a concept that describes the level of decomposition of the functionalities of a program into separate smaller programs (Vera-Rivera et al., 2021). The opposite of *very granular software* would be a software that is *very coarse.* Considering the fact that Monoliths consist of a single deployable code base it can be argued that Monolithic architecture has the coarsest level of granularity a software can have. On the other extreme we have Microservice architecture with a philosophy to split a program into minimal independent programs (Dragoni et al., 2017).

This thesis proposes a more formal way of describing the granularity of a software where granularity is equaled to *one divided by the number of independent units of code.*

$$G = \frac{1}{n} \mid \; G \;\; \textit{is the granularity of a software with} \;\; \boldsymbol{n} \;\; \textit{number of independent units of code}$$

This way the granularity of Monoliths is always *one*, since they per definition consist of *one unit of code.* The granularity of a Microservice-based system follows, *1/n | n= number of microservices.*

To give a better intuitive understanding of the difference in basic structure between Monolithic- and Microservice architecture a visual representation is given in Figure 2.1. More specifically Figure 2.1 visualizes two pieces of software that are functional counterparts in that they both offer the same functionality. The functionalities are *weather forecasting, news*, and a *discussion forum.* In the Monolithic version of the application, we can see that all functionality is within one box, representing *one unit of code.* On the other hand, the Microservice-based version of the program has the functionalities in separate boxes, representing the individual microservices. The lines between the individual microservices represent the communication lines that are needed to combine the individual microservices into a coherent system. Using the formula given for calculating the granularity, the granularity of the two pieces of software seen in Figure 2.1 would be a granularity of *one* for the Monolith, and a granularity of 1/3 for the Microservice-based system.

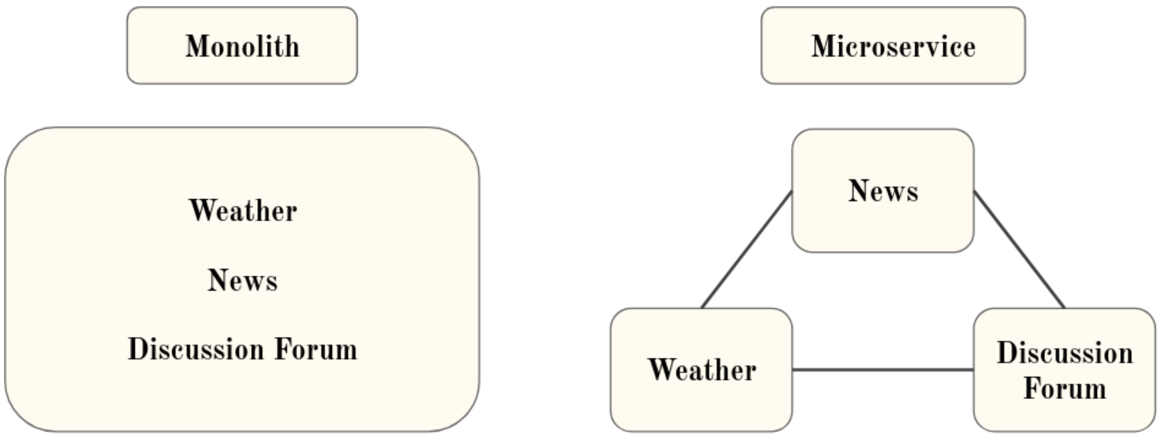


Figure 2.1: A visual representation of two functional counterparts where one is based on Monolithic architecture and the other on Microservice architecture.

3 Monolithic- vs. Microservice Architecture

The quality attributes that have been chosen for the comparison of Monolithic- and Microservice architecture are *scalability, performance, deployability*, and *security.* Considering that there are countless quality attributes to choose from, as displayed by the long lists of quality attributes upheld by the International Organization for Standardization (2023) and Wikipedia (2023), there is no way of incorporating all aspects into the comparison. The motivation for using the *chosen* quality attributes is twofold. Firstly, they are in some way important to the context of comparing Monolithic- or Microservice architecture. Second, they highlight different aspects of software and software development so that the concerns of key stakeholders can be taken into consideration.

The key stakeholders of a software are considered to be the *end-users*, *developers*, and *business owners.* These have been extracted from the stakeholder categorization of Preiss and Wegmann (2001), with the exception of merging the *developers* with *operators.* Developers are those who implement new features into software, while operators are those who handle the tasks related to the software in its execution environment. The merger between developers and operators is motivated by the fact that modern software development often follows *DevOps* practices (Chen, 2018). That is traditionally *development* and *operation* were handled by separate teams, but nowadays it is common for the same team to handle both aspects.

Due to the complex relations between certain quality attributes the structure of the analysis follows a more natural division of chapters, instead of dividing strictly by quality attribute.

3.1 Performance and Scalability: Two Sides of the Same Coin

Performance is defined as a system's ability to meet timing requirements and can be measured by metrics such as how many *requests/second* a software can handle, or through *response time* for a request (Bass et al., 2003). The performance and especially *response time* of a software is key for end-user satisfaction as poor performance pulls the user out from an immersive experience (Doherty and Sorenson, 2015). Analyzing performance in Monoliths and Microservices thus lends us the perspective of the *end-user*. There is inconsistent evidence on how Monoliths and Microservice-based systems perform (Blinowski et al., 2022), (Al-Debagy and Martinek, 2018). Additionally, *performance* has been identified as the second most analyzed quality attribute of Microservice architecture (Li et al., 2021). The combination of giving us a key stakeholder perspective and being relevant to the Monolith-Microservice discussion certifies its place as a quality attribute through which to compare Monolithic- and Microservice architecture.

Scalability is defined as *a software's ability to handle an increasing amount of work* (Kazman and Kruchten, 2012). In other words, it is a software's ability to keep consistent *performance* as the amounts of requests increases. It could thus be viewed as an aspect of *performance*, but at the same time, a trade-off between *scalability* and *performance* has been identified where the granularity of a software plays a key role (Klock et al., 2017). This poses an interesting dynamic to analyze considering that Monolithic- and Microservice architecture make out the two extremes on the spectrum of granularity. Additionally, *scalability* has been identified as the most analyzed quality attribute in the context of Microservice architecture (Li et al., 2021). From a stakeholder perspective, *scalability* could be argued to be important to the *end-user* by considering *scalability* as an aspect of performance.

Communication between individual microservices has been argued to be a factor leading to worse performance in Microservice-based systems than in Monoliths (Richards, 2015). This is due to the fact that the communication between individual microservices happens *over-the-network*, which always will be slower than the purely *local* communication found within Monoliths. To clarify, Monoliths do use network-communication, but all application *internal* communication is local, while application internal communication in

Microservice-based systems to a large degree happens *over-the-network*. This leads to finer granularity having a causal relationship with *worse* performance as the more decomposed a program is, the more over-the-network communication has to be present which can become a bottleneck for performance. Theoretically, this argument is sound, but at the same time, several studies have shown that more granular software can actually perform better when faced with higher workloads, due to being able to scale better (Blinowski et al., 2022), (Gos and Zabierowski, 2020).

Scaling software can be achieved through either *vertical-* or *horizontal scaling*. Vertical scaling refers to allocating more resources (CPU, memory, storage) to existing units of software (Blinowski et al., 2022). The resources are either allocated from the machine that the software is running on or through running the software on a more powerful machine. *Horizontal scaling* refers to increasing the number of physical machines on which a software runs (Blinowski et al., 2022). Further, scaling software horizontally entails *Horizontal Duplication*; a technique for adding and subtracting duplicate instances of the same software. Duplicate instances of a software run on separate machines and the number of duplicates can be regulated in real-time according what is needed (Li et al., 2021). A drawback of Horizontal Duplication is that it leads to overhead in the form of organizing work amongst duplicates and additional infrastructure needed for doing this (Li et al., 2021). Since there is a limit to how powerful computers are available, there is also a limit to vertical scaling (Blinowski et al., 2022). This is also evidenced by the fact that all of the top 500 supercomputers in the world use either a cluster- or massive parallel architecture (*TOP500 Supercomputer Sites* 2023). Both architectures are combinations of several lesser computers. An analogy for vertical- and horizontal scaling could be, using *one big truck* vs. *several smaller trucks*. The big truck can carry more weight than a single small truck but to a limit. The smaller trucks can together carry more than the single big truck, but you need more drivers for this, representing the overhead.

Gos and Zabierowski (2020) have conducted a quantitative analysis of *performance* differences between Monoliths and Microservices with a focus on *requests/second* and *response time*. Though these are related, this thesis focuses on *response time* to capture the *end-user* perspective better. Gos and Zabierowski (2020) compare a Monolithic software vs. a Microservice software which is horizontally duplicated 1-4 times. The study concludes that when faced with higher amounts of requests (300 000 at once) one instance of the Microservice program responds about 23 percent quicker than its Monolith counterpart.

When duplicating the Microservice four times it outperforms the Monolith by 31 percent due to being able to distribute the workload more efficiently. On the other hand, when the Microservice-based system is faced with lighter workloads (30 000 requests at once) the Monolith has about 32 percent better response time than a single instance of the Microservice software. The same, but against four instances of the Microservice software shows a 59 percent difference in favour of the Monolith. The key finding however can best be seen by observing Figure 3.1, taken from (Gos and Zabierowski, 2020). Here we see a converging pattern in the form of declining performance when the number of instances increases and the number of requests stays low and increased performance as the number of instances increases when the number of requests is high. In other words, when the workload is lower the Monolith performs much better than any of the Microservice-based systems. On the other hand performance at higher workloads is better handled by the Microservice-based software.

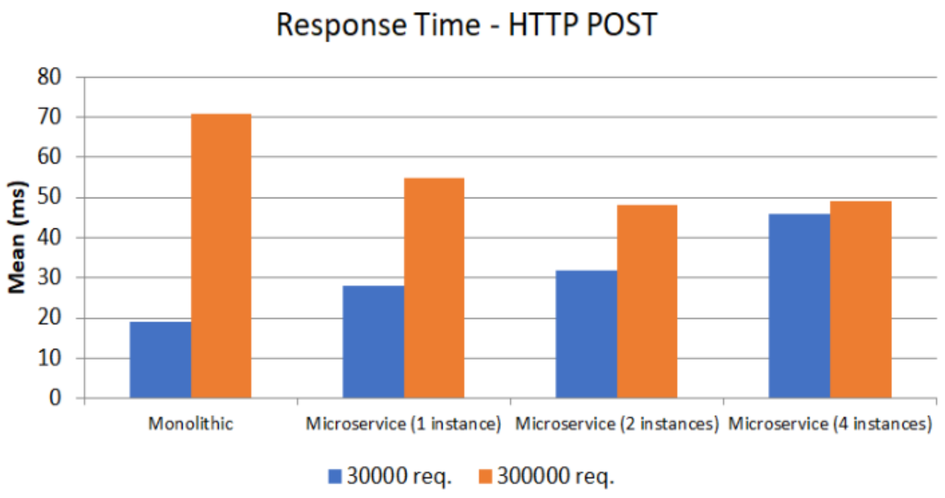


Figure 3.1: The relative performance of the two extremes depends on the number of requests (Gos and Zabierowski, 2020)

.

The converging pattern is however not entirely true. Four instances of the Microserivce-based system actually perform 1 millisecond worse than two instances in the case of (Gos and Zabierowski, 2020). This might be a statistical tie, but Blinowski et al. (2022) point out that their tests showed increased performance with up to *three* horizontally duplicated instances, but adding a *fourth* instance actually worsened performance. The explanation given for this is that the overhead of distributing the workload to more than three instances consumes more resources than it saves. How applicable these results are to other situations remains inconclusive, as there are several circumstantial factors that could lead to different results. Examples of such factors are using a public network and public cloud technology as well as the specific size of the applications used in their tests.

Another *performance*-related aspect of Monolithic- and Microservice architecture that this thesis covers, is the latter's option of *heterogeneous technologies.* Theoretically, there is a big difference between what technology is used. For example, *compiled* programming languages like C++ are usually more performative than *interpreted* programming languages such as Python (Kwame et al., 2017). *Compiling* entails translation of programming language code into machine code, *before* being run. *Interpreted* languages are translated while the code is in execution. Interpreted languages are slower due to translating being part of the execution environment. Another example is the differences in speed between classical relational databases and document-based databases with the latter possessing faster speed (Nayak et al., 2013). This thesis has not found large-scale scientific studies on how the use of heterogeneous technologies affects performance in Microservice-based systems, but Chen (2018) reports that they have made use of different technologies in different microservices to achieve better performance.

All in all *performance* of Monolithic- and Microservice architecture is tightly linked to the circumstances. Going back to the earlier example of a *big truck vs. many smaller trucks* it could be explained like this: One big truck might be faster if it can carry everything in one trip. When the cargo exceeds the limit of the big truck, however, it might actually be faster to use several smaller trucks.

3.2 Efficiency of Scaling

Scaling costs money through the acquirement of more computing power (Blinowski et al., 2022). *Scalability* can thus also be viewed as a quality attribute that is of interest to the key stakeholder of the *business owner.*

According to the calculations of Blinowski et al. (2022) vertical scaling is more cost-efficient than horizontal scaling. Considering that vertical scaling caps at some point, the cost of horizontal scaling is still imperative and can't be ignored. The key difference between Monolithic- and Microservice architecture and *scalability* is that the granular design of Microservice architecture lets it be much more cost-efficient when scaling horizontally.

Efficiency is achieved by being able to independently scale only those individual microservices that are experiencing increased amounts of requests (Fowler and Lewis, 2014). Going back to the example from Chapter 2.2, of the website which offers news, weather forecasts, and a discussion forum, we could imagine a scenario where there is a storm approaching, leading to increased requests for the weather-forecast part of the program in specific. Assuming the workload exceeds the ability of vertical scaling, the software needs to be scaled horizontally. Here the difference becomes apparent. When duplicating a Monolith, we always need to duplicate the whole program, though it's only the weather-forecast service that is in need of the added resources. The unnecessary duplication of the news service and the discussion forum is thus a waste of resources. Using Microservice architecture it is possible to only run duplicates of the weather microservice, solving scalability much more efficiently.

A more formal way to describe the resources saved by only scaling those individual microservices that are in need of it would be the following:

$E = \left(1 - \frac{m}{n}\right) \times 100 \mid$ ***E*** *represents the resources saved in relation to scaling the entire program, where* ***m*** *is the number of microservices requiring scaling, and* ***n*** *is the total number of microservices.*

This formula builds on the formula given for describing the general granularity of some specific software in Chapter 2.2. Horizontal duplication of all parts of an application leads to a 100 percent increase in resource use. Naturally horizontally duplicating a subset saves the resources of the individual microservices that were *not* duplicated. The fact that a Microservice-based system might have up to thousands of individual microservices would give us an example where scaling one out of a thousand individual microservices, could be 99.9 percent more efficient than its Monolith counterpart. The formal description of resources saved is an oversimplification of reality as individual microservices might not be of equal size and may need different amounts of resources based on what the service *does*. Anyhow, it gives a good insight into how profound of an impact the difference in granularity can have on a software and how this translates to a great gap in *scalability* between Monolithic- and Microservice architecture.

Apart from the efficiency of scaling, another aspect to consider is the amount of *time* it takes to scale. The fact that a Monolithic version of the program is much larger than individual microservices makes it much slower to duplicate (Fowler and Lewis, 2014). If comparing a Monolithic- and Microservice-based software of similar size we can again apply the concept of granularity to get a basic measure of the difference in time needed to fire up new duplicates of a program. This is important to consider in the context where a software needs to adapt to rapidly increasing or *decreasing* amounts of concurrent users.

Amazon Elastic Compute Cloud (EC2) is a service that lets you rent compute/server capacity with on-demand pricing. On their web page, they list solutions of different capacities according to the customer's specific needs. For example, their smallest machine t4g.nano provides 0.5 GiB of memory and up to 5 Gigabit Network performance for 0.0042 dollars/hour of use. It is good for general-purpose applications with moderate CPU usage. The most powerful machine they offer is the u-6tb1.112xlarge, with 6144 GiB of memory and 100 Gigabit of Network performance, for a cost of 58.045 dollars/hour of use. It is purpose-built for running large in-memory databases. The specific prices mentioned are from 11.12.2023 and are subject to change (Amazon-Web-Services, 2023) .

Amazon's solution fits like a glove for Microservice architecture, due to such software being able to make use of *heterogeneous technologies*. An individual microservice that is mostly focused on basic tasks could make use of the t4.nano. If the capacity of the server is overwhelmed a horizontal duplicate can be fired up automatically (Amazon-Web-Services, 2023). Another part of the Microservice system, which is focused on demanding tasks, could instead make use of the u-6tb1.112xlarge machine. Comparing this to a functional counterpart using Monolithic architecture highlights the difference in efficiency. First and foremost the Monolith needs to compromise about which solution it runs on. Second, since it always needs to be duplicated as a whole, the possibility of incrementally (and efficiently) increasing capacity with cheap t4.nanos is not possible. Instead, it needs to duplicate the entire application running another instance of a more powerful and costly platform.

3.3 Deployability

Deployability can be defined as the time to get code into production after a commit (Bass, 2016). That is, how long does it take for an update to reach the end user. Deployment of either a single code base or multiple independent programs is central to defining and separating Monolithic- and Microservice architecture. There seems to be a lot of nuance as to whether Monoliths or Microservice-based systems are easier to deploy. Several studies state that Monoliths are easier to deploy (Gos and Zabierowski, 2020), (Blinowski et al., 2022), (Richardson, 2014). Other say that Microservice architecture is better for deployability (Richardson, 2014), (Richards, 2015), (Li et al., 2021), (Blinowski et al., 2022). From a stakeholder perspective, *deployability* most directly affects the developers/operators of software. But how fast a software can be updated can also be considered to be a *business-* and *user interest.*

Richardson (2014) and Blinowski et al. (2022) explain the core of why Monolithic- and Microservice architectures are good in different ways when it comes to deploying software. They argue that Monoliths are simple to deploy in that all you have to do to get an update into the hands of the end-user, is to deploy the one unit of code and see that it works properly. While deployment is simple for small to medium applications, larger Monoliths will prove to contain several issues related to deployment. Firstly, deploying a large Monolith involves several steps of which any might fail, leading to deployment having to start over. Additionally from it being risky and complex, it is also time-consuming. This is time when the program is not in production, but developers can't continue development to a large degree since they are still waiting for the last update to go into production. All of this has to be repeated each time even a small part of the Monolith is updated, as it always needs to be deployed as a single unit. This results in deployments usually being done much less often (Chen, 2018).

Microservice architecture solves many of the above-mentioned problems through its granular design (Richardson, 2014), (Blinowski et al., 2022). By splitting a program into several smaller programs, which can be deployed separately, you avoid the problems of risky, time-consuming, and complex deployments. Chen (2018), confirms many of the claims made by Richardson (2014) and Blinowski et al. (2022) in a report about their experiences in a global company that during a four-year period migrated from using a Monolithic architecture to using Microservice architecture. The report also goes into more detail about how Microservice architecture enables them to reach *zero downtime* between deployments. That is when updating the software, there is no time when end-users can't access the application. One of the main reasons that it's hard to reach zero downtime between deployments in Monoliths is that changes to large databases with complex interrelationships are not feasible to automatize. When such large databases are split up among multiple microservices automatising becomes a reality.

However, it should be noted that Microservice architecture is no silver bullet as it comes with its own set of challenges related to deployment (Richardson, 2014), (Blinowski et al., 2022), (Chen, 2018). While it is true that the smaller size of individual microservices makes it easier to deploy them, this is only true for entirely local changes (Chen, 2018). As soon as changes start to affect other microservices (which is unavoidable), an entirely new aspect of complexity in the form of integration is introduced. Chen (2018) explains how the newfound speed at which deployments were made and the challenge of integrating these deployments into the larger program was impossible to handle manually. To handle the new challenges of deployment, they had to build an entirely new custom software that could handle deployment automatically. Such a software absolutely did not come for free as it was developed by a team of eight experts over four years; a total of 32 person-years of effort. Considering this means that, to actually reap the benefits of Microservice deployment you also need to invest a lot which might not be feasible in all situations.

Lastly, the difference in granularity between Monolithic- and Microservice architecture and the fact that individual microservices can be deployed independently of each other can have a major impact on how a business may be structured. Microservice architecture lets a business organize teams along the lines of the individual microservices of a Microservice-based system (Sha Ma, 2022). This can lead to faster decision-making, a better understanding of what the teams do, easier hiring, and more. Decisions can be made faster since individual microservices can be changed independently without affecting the larger program. Understanding of what teams do is easier since the structure of the business follows the structure of the architecture of the software. That is the software- and enterprise architecture mirror each other. Hiring is easier to do since there is a broader pool of technologies that can be used in the program due to being able to use *Heterogeneous technologies*. Monolithic architecture is not as well suited for this form of organizing as the architecture is much more centralized which also needs to be reflected in the business structure.

3.4 Security

Security is defined as a system's ability to protect data from unauthorized parties while retaining access to people who *are* authorized (Bass et al., 2012). *Security* is used to compare Monoliths and Microservices on the basis of three main arguments. First and foremost security itself is a most pressing issue as the number of cyber-crimes rises rapidly

(Sarre et al., 2018). Secondly, Monoliths and Microservices have central differences between them which have major implications on *security*, with both benefits and challenges on both sides (Mateus-Coelho et al., 2021). Third, incorporating *security* in the comparison contributes an additional viewpoint from which to evaluate a software architecture. The stakeholders most interested in *security* can be argued to be the *end-user*, but also *business owners* have a vested interest in the reputation of their company.

Let's first deal with the aspect not present at all in Monolithic architecture; *internal over-the-network communication*. Since Monoliths are deployed as one unit they always run on a single computer. The Monolith might be communicating with external parties, but this is not on the same scale as the *internal* over-the-network communication present in Microservice-based systems. Figure 3.2 visualizes the difference in the area of attack present in the two architectures by marking it in red. Here we can see that the attack area present in Monolithic architecture consists of attacks coming from the outside. On the other hand, we can see that present in the Microservice-based system, there are also communication lines between the individual microservices.

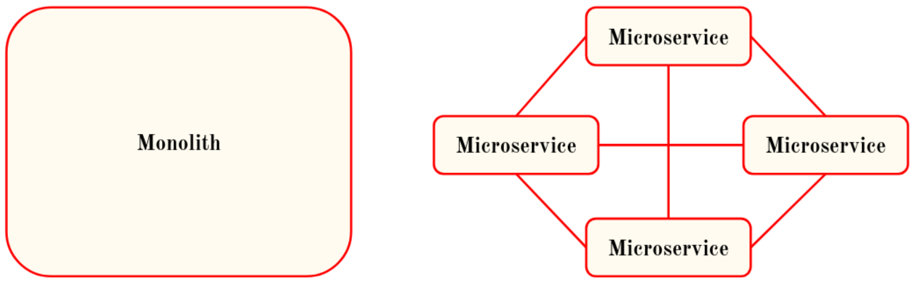


Figure 3.2: The area of attack present in Monoliths and Microservice based systems is marked in red.

Securing these communication lines is complex and risky due to relying on complex technical implementations (Mateus-Coelho et al., 2021). This complexity shows itself in many stages. The development of secure services in itself is a challenge. Additionally, monitoring and debugging a distributed system is on another level compared to Monoliths (Mateus-Coelho et al., 2021). The area of attack made out of these *internal*, yet *over-the-network* communication lines, poses a risk where an attacker can compromise and take control of one of the individual microservices in a software and use its privileges for various malicious intent (Mateus-Coelho et al., 2021). This leads to one of the core security issues in Microservice architecture; *individual microservices can't be trusted.* This fact means that any Microservice-based system needs to be untrusting of its individual microservices, while a Monolithic system can focus its attention on the much narrower attack surface of attacks from the outside. On top of it being technically hard to secure Microservices, there is also a lack of research regarding information security in Microservices, which further complicates securing such systems (Mateus-Coelho et al., 2021).

Distributed denial-of-service is a form of cyber-attack that sets out to crash the server on which some program is running by overloading it with requests (Chung, 2012). Considering that Microservice-based software is much more flexible at handling *scalability* it is also much better positioned to handle such attacks. In this view *distributed denial-of-service* attacks are also an issue of *scalability* (Chung, 2012). Though better *scalability* might help with the issue of servers crashing, increased demand for servers is still a cost issue for businesses and can thus not be ignored. The worst symptoms can however be mitigated with Microservice architecture. Another aspect related to *distributed denial-of-service* attacks is that Microservice architecture lets the system fail more gracefully than a Monolith. What is meant by this is that if any part of a Monolith fails, the whole system goes down with it. A Microservice-based system, on the other hand, can crash partially, with only some of the individual microservices being unavailable (Yarygina and Bagge, 2018).

Monoliths that have been developed over an extended amount of time tend to be hard to modify where changes in one place lead to cascading errors across the software (Blinowski et al., 2022). In other words, large Monoliths can be harder to update, meaning security vulnerabilities are also harder to fix. On the other hand, Microservice-based systems are able to be updated faster as long as the changes are local to individual microservices (Chen, 2018). Thus it can be argued that security vulnerabilities are also faster to fix in Microservice-based systems. Another aspect of updating software is updating the third-party libraries used by a software. According to (Kula et al., 2018), over 80 percent of software-systems tend to keep outdated dependencies, and security-related vulnerabilities found in such dependencies make out a serious security threat. Here the concept of *heterogeneous technology* again plays a role. If a library shows to have security-related vulnerabilities it is much easier to swap it out from individual microservices due to the fact that it doesn't affect the larger program. That is, *Microservice architecture leads to smaller dependency trees.* On the other hand, swapping out a library used in several parts of a Monolithic system is an example of the challenges mentioned by Blinowski et al. (2022) that can lead to cascading errors across the system.

The last security aspect that this thesis highlights is that of *isolation* within an application. The fine granularity Microservice based software allows for better isolation of an attack (Yarygina and Bagge, 2018). What is meant by isolation in this case is that, since individual microservices are their own units of software they can be isolated, killed, and replaced, in case an intruder inflicts a service. This is not possible to the same degree in a Monolith where the program works in one memory address space. Shutting the Monolith down also entails that the whole system goes down which means *end-users* lose access to it.

4 Results

The comparison of Monolithic- and Microservice architecture through the lens of *performance, scalability, deployability*, and *security* has shown that there are strengths and weaknesses in both architectures. The reasons behind these strengths and weaknesses are often multi-faceted and dependent on several situational parameters. To be able to make some conclusions about the specific use-cases for Monolithic- and Microservice architecture there needs to be a view of the big picture from which overarching themes about the two architectures can be identified.

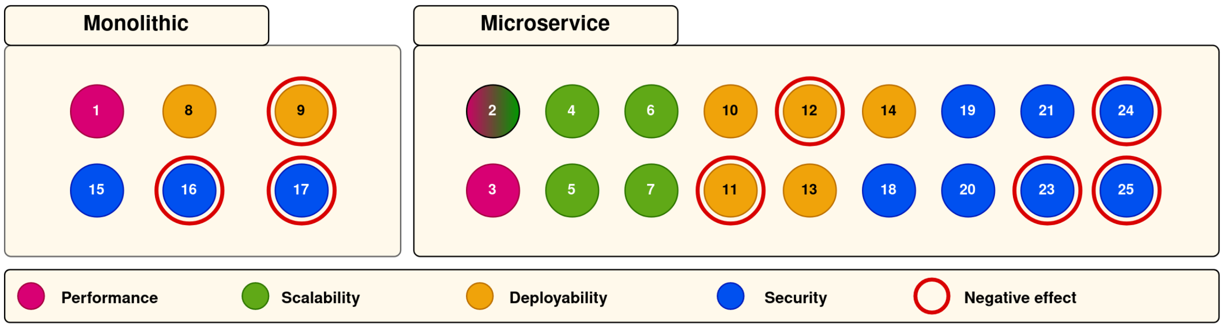
4.1 25 traits of Monolithic- and Microservice architecture

To view the big picture of the differences between Monolithic and Microservice architecture the results of the analysis performed in Chapter 3. are presented as a set of 25 traits adhering to Monolithic- and Microservice architecture. The traits consist of concepts that either lead to differences or are the direct result of using Monolithic- or Microservice architecture. From these traits, conclusions about specific use-cases can be extracted.

The traits are presented in Figure 4.1 which organizes them as belonging to either Monolithic- or Microservice architecture. Due to most traits expressing dynamics with opposite effects in the two architectures, the traits are expressed mostly on the side where they "matter more". This is to avoid a lot of repetition like *"Monoliths have less..."* and another, mirroring trait *"Microservices have more..."*. This is not absolute, however, as some traits are expressed on both sides if they are especially important.

The traits are strongly related to quality attributes and to express this relation the traits are colour-coded accordingly. If the quality attribute is not clear, a combination of colours is used. Additionally, there will be a red ring marking negative traits. Organizing traits like this, gives a chance of viewing everything at once, leading to a better understanding of the big picture.

From the figure, we can immediately see that the majority of traits are on the side of Microservice architecture. The reason for this might be multi-faceted. First and foremost it may be due to the way the traits are categorized (avoiding repetition). Second, this thesis has observed that scientific articles seldom have an exclusive focus on Monolithic architecture. Almost all articles either focus solely on Microservice architecture or both architectures. Monolithic architecture is the older, traditional architecture, while Microservice architecture is newer (Blinowski et al., 2022). This might explain the difference in exclusive focus. That is, more interest is found where there is something new and contrasting to analyze. Before Microservice architecture was introduced there wasn't such a clear counter part to compare Monolithic architecture to. The third reason might be that Microservice-based systems consist of several smaller programs that on their own essentially are small *Monoliths*. This means that many of the theoretical concepts present in Monolithic architecture, are indirectly also present in Microservice architecture. That is, concepts present in Monoliths are present in Microservices, but not the other way around.



Monolithic Architecture	Microservice Architecture	
1. Monoliths perform better at low workloads	2. Microservices perform better at high workloads	14. Zero downtime when deploying updates
8.When small, Monoliths are simpler to deploy	3. Heterogeneous technologies may improve performance	18. Microservices can be more robust against Ddos attacks
9. Big Monolith deployments are slow and risky	4. Independent scaling of microservices	19. Microservices can fail gracefully
15. Monoliths have less entry points that need securing	5. Potentially 1-1/n more efficient scaling	20. Heterogeneous --> switch dependencies faster
16. Big Monoliths can have hard to replace dependencies	6. Heterogeneous machines --> cost benefits when scaling	21. Microservices have smaller dependency trees
17. If a Monolith crashes it is usually total	7. Microservices can react fast to rapidly increasing or decreasing workloads	22. Better isolation between parts of the program
	10. Business can structure its organization along independently deployable microservices	23. Lack of research about security in Microservices
	11. Integration between microservices when deploying non-local changes can be a challenge	24. Microservice communication lines need to be secured
	12. Deploying microservices needs extra infrastructure to realize benefits	25. Individual microservices can't be trusted
	13. Microservices can be deployed faster	

Figure 4.1: 25 traits identified in the analysis are categorized by quality attribute and architecture. Negative effects are marked with a red circle.

We can study the figure to see some patterns formed by the individual traits. First, we can see that two of the three positive traits [1,8] for Monolithic architecture, are related to smaller applications. The negatives continue this pattern where two out of the three negative traits [9, 16] are related to situations where Monoliths grow big. A striking weakness of Monolithic architecture can be seen in the lack of any positive traits about *scalability*. That is, all of the *scalability* traits [2, 4, 5, 6, 7] are found on the side of Microservice architecture.

On the side of Microservice architecture, we can observe the largest found pattern pattern in traits [3, 4, 5, 6, 7, 10, 13, 14, 18, 19, 20, 21, 22]. All of these have in common that it is the finer granularity of Microservice architecture that leads to these benefits. Within this pattern, we see in traits [3, 6, 20] that the possibility of using *heterogeneous technologies* in Microservice-based systems can lead to benefits on several fronts. A negative pattern can be seen in traits [12, 24], that display how Microservice architecture needs additional infrastructure to function properly. Another pattern is that four out of the five negative traits [11, 12, 24, 25] of Microservice architecture are constituted by practical challenges, rather than theoretical limits. This means that as long as these practical challenges can be solved most of the negatives of Microservices are eliminated.

Outside of these architecturally internal patterns, we see the fact that granularity has an opposite effect on performance in traits [1, 2]. Trait [2] also displays how *performance* at high workloads is linked to *scalability*. In traits [17, 19] we see how granularity affects failures in Monolithic- and Microservice-based systems. Trait [23], is different from the others in that it displays a meta-challenge present in building Microservices.

All in all, it can be stated that the overarching themes for Monolithic- and Microservice architecture stem from the core structure of each architecture. More specifically, the difference in granularity between Monolithic- and Microservice architecture lies at the heart of many of their strengths and weaknesses. This pattern shows that the granular design of Microservice architecture gives more granular control of *performance, scalability, deployability*, and *security*. At the same time, finer granularity increases complexity by demanding more granular decisions. That is *the possibility of control, demands control.* On the other side of the spectrum of granularity, Monolithic architecture, lacks this detailed control, but instead expresses simplicity. That is, *with less control comes less responsibility.*

4.2 Use-cases of Monolithic- and Microservice architecture

When choosing between the Monolithic- and Microservice architecture, the latter seems to be much more versatile, being able to create tailored solutions on an individual microservice basis. At the same time the challenges that Microservice architecture introduces in the form of managing a distributed system with additional infrastructure, need to be dealt with for the benefits to be realized. Additionally, it should be noted that most of the negatives related to Monolithic architecture are only present if the Monolith constitutes a big system. This thesis makes three main conclusions about use-cases for Monolithic- and Microservice architecture:

The first main conclusion about use-cases is:

While a system remains small the benefits of Microservice architecture are not realized and the downsides of Monolithic architecture are not present. Thus, for a small system, Monolithic architecture is probably better. When a system grows larger, Microservice architecture is probably better.

The second main conclusion about use-cases is:

If a system needs extreme scalability in some manner, Monolithic architecture might simply not be able to handle it. Thus Microservice architecture should be used.

The third main conclusion about use-cases is targeted at understanding the practicalities:

The practical challenges present in Microservice architecture need to be taken into consideration. If the developer team of a software is not skilled with the relevant technology or doesn't have time to learn such technologies, Monolithic architecture might be a better choice.

When it comes to choosing the right architecture, what it boils down to is identifying the demands put on your software and understanding the capabilities of the developers of the software.

4.3 Validity of the Results

This thesis set out to compare Monolithic- and Microservice architecture from a broad set of stakeholder perspectives. Using quality attributes as a lens through which to analyze software architecture has brought forward several contrasts between the two software architectures. Additionally, the chosen quality attributes have brought forward concepts and dynamics that lie in the interest of all stakeholders. *Performance* (and *Scalability*) in the form of *response time* has touched on how *user satisfaction* could differ between Monoliths and Microservices. The *efficiency of scaling* software brought forward differences in *cost* which is especially important to the *business owner*. *Deployability* captured how the difference in architecture can have significant effects on the practices that *developer/operator* teams use. Additionally, the contrasts between how software is deployed show aspects that are of high interest to *business owners*. *Security* resulted in several observations that are relevant to both *end-users* and *business owners*. Considering that the quality attributes have brought forward aspects relevant to all the stakeholders in focus, the internal validity can be considered at least reasonably high.

Considering that there are countless other quality attributes that could be used to compare Monolithic- and Microservice architecture, it is hard to know if the results of this thesis actually reflect the big picture. The quality attributes that *were* chosen, were so on the basis of being relevant to the discussion about Monolithic and Microservice architecture (and reflecting the interests of stakeholders). The quality attributes have all been identified by several studies as some of the most important quality attributes in the Monolithic- vs. Microservice architecture discussion. This is better than choosing other less relevant quality attributes, but due to the scale of other attributes left out, the are still open questions. Additionally, the quality attributes that were used could have been further analyzed from other perspectives using different metrics. Due to the scope and focus of this thesis, these were however left out.

When it comes to the findings of the thesis there is a heavy focus on *the size of the software* and how this might be the deciding factor in what architecture makes sense to use. This still leaves the open question of *what constitutes small or big software*, which leaves the results quite open for interpretation.

5 Conclusion

This thesis has brought forward some of the most striking differences, as well as some important nuances between Monolithic- and Microservice architecture. To do this it first defined some key concepts important to the topic at hand. Secondly, it explained why Monoliths and Microservices make an interesting comparison due to constituting the two extremes on the spectrum of granularity. Third, it explained the logic behind choosing specific quality attributes to use for comparing the two architectures. After this, the actual comparison was made by analysing Monoliths- and Microservices through the lens of the chosen quality attributes. The results of the analysis were gathered as a set of *25 traits* that were presented in Figure 4.1 which aims to capture the big picture of how Monolithic- and Microservice architecture differ. The results were then discussed to be able to identify some overarching themes and use-cases between Monolithic- and Microservice architecture. Lastly, the validity of the results was analyzed.

The main findings of this thesis are the following. First and foremost this thesis has shown that the core structural differences in Monolithic- and Microservice architecture lead to a set of strengths and weaknesses. More specifically, this thesis has shown how the difference in granularity leads to specific overarching themes for each architecture. The core difference is that the granular design of Microservice architecture allows for greater control of a software, but adds complexity. That is *the possibility of control, demands control.* Monolithic architecture which displays the least granular design possible, doesn't allow for such detailed control but instead expresses a simpler design with less aspects to manage. That is, *with less control comes less responsibility.*

This thesis has also shown how Monolithic- and Microservice architecture each has its own use-cases. The simplicity of Monolithic architecture makes it a good choice for smaller applications. Additionally, most of the negative sides of using Monolithic architecture only appear when a system grows very big. On the other hand, the detailed control that Microservice architecture allows for might be very useful in *big systems* but might be unnecessary in smaller systems. Additionally, this thesis has shown that Microservice architecture is vastly superior to Monolithic architecture when it comes to *scaling* software.

Finally, the results of this thesis have brought forward subjects for future research. Firstly, *How do Monolithic- and Microservice architecture differ in other quality attributes than those used in this thesis?* Second, *how do Monolithic- and Microservice architecture differ when using other metrics/aspects of performance, scalability, deployability, and security?* Third, *what constitutes a small or a big software system?*

Bibliography

Amazon-Web-Services (2023). *Amazon EC2 Pricing*. Accessed: 9.12.2023. URL: https://aws.amazon.com/ec2/pricing/on-demand/.

Bass, L. (2016). "Deployability". In: *Software Quality Assurance*. Ed. by I. Mistrik, R. Soley, N. Ali, J. Grundy, and B. Tekinerdogan. Boston: Morgan Kaufmann, pp. xxiii–xxvii. ISBN: 978-0-12-802301-3. DOI: https://doi.org/10.1016/B978-0-12-802301-3.00019-3. URL: https://www.sciencedirect.com/science/article/pii/B9780128023013000193.

Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

– (2012). *Software Architecture in Practice (Third Edit., p. 624)*.

Blinowski, G., Ojdowska, A., and Przybyłek, A. (2022). "Monolithic vs. microservice architecture: A performance and scalability evaluation". In: *IEEE Access* 10, pp. 20357–20374.

Carnegie-Mellon-University-Software-Engineering-Institute (2017). *What Is Your Definition of Software Architecture?*

Chen, L. (2018). "Microservices: architecting for continuous delivery and DevOps". In: *2018 IEEE International conference on software architecture (ICSA)*. IEEE, pp. 39–397.

Chung, Y. (2012). "Distributed denial of service is a scalability problem". In: *ACM SIGCOMM Computer Communication Review* 42.1, pp. 69–71.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond*. 2nd. Addison-Wesley.

Al-Debagy, O. and Martinek, P. (2018). "A comparative review of microservices and monolithic architectures". In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. IEEE, pp. 000149–000154.

Doherty, R. A. and Sorenson, P. (2015). "Keeping users in the flow: mapping system responsiveness with user experience". In: *Procedia Manufacturing* 3, pp. 4384–4391.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering*, pp. 195–216.

Egyed, A. and Grunbacher, P. (2004). “Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help”. In: *IEEE Software* 21.6, pp. 50–58. DOI: 10.1109/MS.2004.40.

Fowler, M. and Lewis, J. (2014). “Microservices, 2014”. In: *URL: http://martinfowler. com/articles/microservices. html* 1.1, pp. 1–1.

Gan, Y. and Delimitrou, C. (2018). “The architectural implications of cloud microservices”. In: *IEEE Computer Architecture Letters* 17.2, pp. 155–158.

Gos, K. and Zabierowski, W. (2020). “The comparison of microservice and monolithic architecture”. In: *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE, pp. 150–153.

International Organization for Standardization (2023). *ISO 25010*. ISO 25010 Standards. URL: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

ISO/IEC/IEEE 24765:2017 (2017). Tech. rep. International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), Institute of Electrical and Electronics Engineers (IEEE). URL: https://www.iso.org/standard/71952.html.

Kazman, R. and Kruchten, P. (2012). “Design approaches for taming complexity”. In: *2012 IEEE International Systems Conference SysCon 2012*. IEEE, pp. 1–6.

Klock, S., Van Der Werf, J. M. E., Guelen, J. P., and Jansen, S. (2017). “Workload-based clustering of coherent feature sets in microservice architectures”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, pp. 11–20.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018). “Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration”. In: *Empirical Software Engineering* 23, pp. 384–417.

Kwame, A. E., Martey, E. M., and Chris, A. G. (2017). “Qualitative assessment of compiled, interpreted and hybrid programming languages”. In: *Communications* 7.7, pp. 8–13.

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., and Babar, M. A. (2021). “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review”. In: *Information and software technology* 131, p. 106449.

Mateus-Coelho, N., Cruz-Cunha, M., and Ferreira, L. G. (2021). “Security in microservices architectures”. In: *Procedia Computer Science* 181, pp. 1225–1236.

Nayak, A., Poriya, A., and Poojary, D. (2013). “Type of NOSQL databases and its comparison with relational databases”. In: *International Journal of Applied Information Systems* 5.4, pp. 16–19.

Preiss, O. and Wegmann, A. (2001). “Stakeholder discovery and classification based on systems science principles”. In: *Proceedings Second Asia-Pacific Conference on Quality Software*. IEEE, pp. 194–198.

Rademacher, F., Sachweh, S., and Zündorf, A. (2019). “Aspect-oriented modeling of technology heterogeneity in microservice architecture”. In: *2019 IEEE International conference on software architecture (ICSA)*. IEEE, pp. 21–30.

Richards, M. (2015). *Microservices vs. service-oriented architecture*. O’Reilly Media Sebastopol.

Richards, M. and Ford, N. (2020). *Fundamentals of software architecture: an engineering approach*. O’Reilly Media.

Richardson, C. (2014). *Microservices: Decomposing Applications for Deployability and Scalability*. Accessed 29 June 2016. URL: https://www.infoq.com/articles/microservices-intro.

Sarre, R., Lau, L. Y.-C., and Chang, L. Y. (2018). *Responding to cybercrime: current trends*.

Sha Ma, B. L. (Sept. 2022). “GitHub’s Journey from Monolith to Microservices”. In: *InfoQ*. URL: https://www.infoq.com/articles/github-monolith-microservices/.

Solms, F. (2012). “What is software architecture?” In: *Proceedings of the south african institute for computer scientists and information technologists conference*, pp. 363–373.

TOP500 Supercomputer Sites (2023). Accessed on: 2023. URL: https://www.top500.org/statistics/sublist/.

Vera-Rivera, F. H., Gaona, C., and Astudillo, H. (2021). “Defining and measuring microservice granularity—a literature overview”. In: *PeerJ Computer Science* 7, e695.

Wikipedia (2023). *List of System Quality Attributes*. Wikipedia. URL: https://en.wikipedia.org/wiki/List_of_system_quality_attributes.

Yarygina, T. and Bagge, A. H. (2018). “Overcoming security challenges in microservice architectures”. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, pp. 11–20.

Appendix A Extended abstract in Swedish

Att välja rätt mjukvaruarkitektur är kritiskt då man utvecklar mjukvarupräglade system, eftersom arkitekturen i stor utsträckning är avgörande i systemets förmåga att uppfylla kraven hos systemets intressehavare. Två prevalenta och mycket använda mjukvaruarkitekturer är Monolitisk- och Mikrotjänstarkitekur. Gemensamt för alla Monoliter är att de i grunden utgörs av en distribuerbar enhet kod. Mikrotjänstarkitektur utgör en motsatt filosofi, var man strävar efter att dela upp ett program i minimala, självständigt distribuerbara små program som genom kommunikation sammanställs till ett enhetligt system. Dessa motsatta filosofier kan också förklaras som att arkitekturerna utgör extremen på kornighetsspektrumet. Med kornighet syftas nivån av hur uppdelat en mjukvara är i mindre enheter. D.v.s. Monoliter utgör den grövsta kornigheten, medan Mikrotjänstarkitektur förespråkar den finaste möjligaste kornigheten.

Denna text strävar efter att jämföra Monolitisk- och Mikrotjänstarkitektur från ett brett perspektiv i syfte av att skapa en helhetsbild över dessa två mjukvaruarkitekturers användningsområden. Ett brett perspektiv definieras som ett perspektiv som beaktar de viktigaste intressehavarna för ett system. Denna text beaktar dessa som slutanvändaren, programutväcklaren och företagsägaren. För att mäta hur väl eller på vilket sätt en mjukvaruarkitektur uppfyller intressehavarnas krav används ett urval av kvalitetsegenskaper. Kvalitetsegenskaper definieras som egenskaper vilka kan användas som indikation för hur väl ett system uppfyller dess intressehavares krav. De valda kvalitetsegenskaperna är prestanda, skalbarhet, distribution och säkerhet. Valet av specifikt dessa kvalitetsegenskaper baserar sig på tre huvudsakliga argument. Ett, de har inom den vetenskapliga litteraturen identifierats att utgöra bland de viktigaste kvalitetsegenskaperna inom Monolitisk- eller Mikrotjänsarkitektur. Två, var och en av kvalitetsegenskaperna reflekterar krav av åtminstone en av de viktigaste intressehavarna. Tre, tillsammans täcker kvalitetsegenskaperna alla av de viktigaste intressehavarna.

Prestanda definieras som ett systems förmåga att uppnå tidskrav och kan mätas genom ett antal olika mått och kriterier. Exempel är hur många förfrågningar/sekund ett system kan hantera, och responstid, d.v.s. tiden det tar från och att en användare av ett program ger en input till att användaren ser resultatet av hens input. Prestanda och speciellt responsetid är kritiskt för slutanvändarbelåtenhet och bidrar därmed en kvalitetsegenskap som reflekterar slutanvändaren som intressehavare. Dessutom har prestanda identifierats som den andramest analyserade kvalitetsegenskapen inom vetenskaplig litteratur om Mikrotjänst arkitektur.

Skalbarhet definieras som ett systems förmåga att hantera ökade mängder arbete. Man kunde även betrakta skalbarhet som förmågan hos ett system att bavara prestandan, även när arbetsmängden stiger. Detta starka förhållande mellan skalbarhet och prestanda betyder att dessa kvalitetsegenskaper kan betraktas som två sidor av samma mynt. Skalbarhet har identifierats som en kvalitetsegenskap med mycket konfliktartad information inom den vetenskapliga litteraturen. Dessutom har skalbarhet även identifierats som den mest analyserade kvalitetsegenskapen inom vetenskaplig litteratur om Mikrotjänstarkitektur. I och med skalbarhets nära förhållande till prestanda kan det argumenteras att det är en central kvalitetsegenskap för slutanvändaren.

Kommunikationen mellan individuella mikrotjänster har framförts som en faktor som leder till sämre prestanda i Mikrotjänstbaserade system jämfört med Monoliter. Det beror på att kommunikationen mellan individuella mikrotjänster sker över nätverket, vilket alltid kommer att vara långsammare än den rent lokala kommunikationen som finns inom monoliter. Detta leder till att finare granularitet har ett orsakssamband med sämre prestanda eftersom ju mer nedbruten en programvara är, desto mer över-nätverkskommunikation måste finnas, vilket kan bli en flaskhals för prestanda. Teoretiskt sett är detta argument konsistent, men samtidigt har flera studier visat att mer granulär mjukvara faktiskt kan prestera bättre vid högre arbetsbelastning.

Skalning av programvara kan uppnås antingen genom vertikal skalning eller horisontell skalning. Vertikal skalning innebär att tilldela fler resurser (CPU, minne, lagring) till befintliga enheter av programvara. Horisontell skalning innebär att öka antalet fysiska maskiner där en programvara körs. Vidare innebär horisontell skalning Horizontal Duplication; en teknik för att lägga till och ta bort duplicerade instanser av samma programvara. Duplicerade instanser av en programvara körs på separata maskiner och antalet kopior kan regleras i realtid efter behov. En nackdel med Horizontal Duplication är att det innebär resursanvänding som går åt till organisering av arbetet istället för själva arbetet. Eftersom det finns en begränsning för hur kraftfulla datorer som är tillgängliga, finns det också en begränsning för vertikal skalning, vilket gör horisontell skalning ett måste efter en viss punkt. En analogi för vertikal- och horisontell skalning kan vara att använda en stor lastbil jämfört med flera mindre lastbilar. Den stora lastbilen kan bära mer vikt än en enskild liten lastbil men har en begränsning. De mindre lastbilarna kan tillsammans bära mer än den enskilda stora lastbilen, men du behöver fler förare för detta, vilket representerar resursförlusten till följd av organiseringen av arbetet.

Skalning påverkar prestanda i den mån av att ökad arbetsmängd kräver mera resurser, vilket förutsätter att programmet skalas antingen vertikalt eller horisontellt. Monoliter och Mikrotjänstbaserade system skilljer sig centralt i den mån av att Mikrotjänst arkitektur är lättare att skala horisontellt. Detta leder till den centrala dynamiken gällande prestanda, närvarande inom Monolit vs. Mikrotjänst diskussionen. Denna centrala dynamik säger att Monoliter har bättre prestanda vid lägre mängder arbete eftersom den lokala kommunikationen är snabbare än kommunikation över-nätverket. Och andra sidan tillåter Mikrotjänstarkitekturens mer effektiva skalning den att distribuera arbetet mellan flera instancer vilket leder till bättre prestanda vid höga arbetsmängder. Denna dynamik kan också formuleras som att kornighet har en motsatt effekt på prestanda vid låga och höga mängder arbete i form av negativ inverkan vid låg mängd arbete och positiv inverkan vid hög mängd arbete.

En annan prestandarelaterad aspekt mellan monolitiska och mikrotjänstarkitekturer är den senarenämdas möjlighet till heterogena teknologier. Heterogena teknologier innebär att individuella mikrotjänster kan använda sig av aldeles olika teknologier. Teoretiskt sett skiljer sig valet av teknik markant. Till exempel är kompilerade språk som C++ oftast snabbare än tolkade språk som Python. Ett annat exempel är hastighetsvariationen mellan traditionella relationsdatabaser och dokumentbaserade databaser, där de senarenämnda är snabbare.

Slutligen kan det konstateras att prestanda för Monolitisk- och Mikrotjänstarkitektur starkt är kopplad till omständigheterna. För att gå tillbaka till det tidigare exemplet med enstor lastbil jämfört med många mindre lastbilar kan förklaras så här: En stor lastbil kan vara snabbare om den kan transportera allt på en resa. När lasten överskrider begränsningen för den stora lastbilen kan det emellertid faktiskt vara snabbare att använda flera mindre lastbilar.

Förutom skalbarhets starka relation till prestanda är det också i högsta grad en kvalitetsegenskap som är av intresse för företagsägare i form av att utgöra en stor kostnadsfråga. Att skala mjukvara kostar i form av att anskaffa sig mera beräkningskraft i form av serverkapacitet. Vertikal skalning har identifierats som mera konstnadseffektivt än horisontell skalning. P.g.a. vertikal skalnings begränsningar är dock även konstader relaterade till horisontell skalning centrala att analysera. Den huvudsakliga skillnaden mellan Monolitisk- och Mikrotjänstarkitektur när det gäller costnadseffektiveten av skalbarhet är den mer finkorniiga designen av Mikrotjänstarkitektur. Mer specifikt gör denna finkorniga design horisontell skalning mycket mera kostandseffektiv. Effektivitet uppnås genom att de individuella mikrotjänster kan skalas självstandit. D.v.s. Enbart de individuella mikrotjänster som upplever ökad arbetsmängd kan ges mera resurser, medan andra delar av programmet förblir orörda. Och andra sidan är detta inte möjligt med en Monolitisk arkitektur var hela programmet utgörs av en enhet kod som alltid måste dupliceras som en helhet. Vi föreställa oss en nätsida som erbjuder nyheter, väderleksprognos och diskussionsforum. Läget är följande: En stor storm närmar sig, vilket leder till ökad använding av specifikt väderleksprognostjänsten. Här blir skillnaden tydlig. När en Monolit dupliceras måste hela programmet dupliceras, även om det bara är väderprognostjänsten som behöver extra resurser. Onödig duplication av nyhetstjänsten och diskussionsforumet är därför en slöseri med resurser.

Distribution kan definieras som tiden det tar för att en updatering av en mjukvara skall nå slutanvändaren. Hur en mjukvara distribueras är centralt för själva definitionen av Monolitisk- och Mikrotjänstarkitektur. Den vetenskapliga litteraturen innehar mycket nyans i diskussionen gällande huruvida Monoliter eller Mikrotjänstbaserade system är lättare att distribuera. Flera studier argumenterar för att Monliter är lättare att distribuera, medan andra menar att Mikrotjäsntbaserade system är lättare att distribuera. Flera studier påpekar även att bägge arkitekurer har sina styrkor och svagheter. När det kommer till intressehavare av distribution, kan det nämnas att distribution, mest direkt påverkar programutväcklare, men även företagsägare och slutanvändaren är inresserade av hur snabbt en mjukvara kan uppdateras.

De centrala koncepten som påverkar distribuering i kontexten av Monolitisk- och Mikrotjänstarkitektur är storleken av program och integration av olika program. Monoliter är enklare att distribuera i den mån av att de utgörs av en enhet kod. Detta innebär att så länge distribueringen lyckas så finns det inte några ytterligare utmaningar i form av integration. Monoliter är därmed lättare att distribuera så länge mjukvaran inte utgör ett mycket stort system. Stora system är i almänhet utmanande att distribuera eftersom det finns så många steg som kan gå fel. Dessutom måste allting börjas från början om ens ett steg går fel. Ytterligare är distribuering av stora system mycket långsamt vilket leder till dödtid för programutveklarna. Detta utgör de stora utmaningarna inom distribuering av stora Monoliter. Varje gång en Monolit skall updateras måste denna tidskrävande och riskfyllda process upprepas.

Mikrotjänstarkitektur löser flera av dessa utmaningar genom sin finkorniga design. Genom att splittra det stora programmet i flera mindre självständiga enheter som var för sig kan distribueras skillt undviker man de utmaningar förknippade med distribueringar av stora system. Detta leder till fördelar så som snabbare distribuering och ingen dödtid. En storskalig fördel är dessutom företagets val att strukturera sin företagsorganisation enligt mjukvarans arkitektur var olika arbetsgrupper fördelas till sina egna individuella mikrotjänster. Detta medför i sin tur fördelar så som snabbare beslutsfattning då den självstandiga naturen hos individuella mikrotjäsnter tillåter beslut att göras internt utan påverkan på andra delar av mjukvaran och företaget.

Mikrotjänstarkitektur för dock med sin egen hop av utmaningar i form av integration mellan individuella mikrotjänster då uppdateringar till en individuell mikrotjänst innebär icke-lokala förändringar. Detta i kombination, med hastigheten som distribuering av Mikrotjänstbaserade system har innebär ofta att ytterligare mjukvaruinfrastruktur behövs. Mera specifikt kräver det ofta att hela distribueringsprocessen automatiseras fullständigt. Skapandet av en sådan infrastruktur kan vara en stor utmaning i sig.

Säkerhet kan definieras som ett systems förmåga att samtidigt försvara data från icke-lovliga entiteter, som systemet ger lovliga entiteter åtkomst till deras data. Först och främst är säkerhet ett kritiskt tema i och med den kraftiga ökningen av cyberbrott. Dessutom innehar Monolitisk- och Mikrotjänstarkitektru centrala skillnader som påverkar säkerhet i stor grad. Säkerhet bidrar dessutom med ytterligare täckning av intressehavarperspektiv i form av att vara speciellt viktigt för slutanvändaren som vill använda cybersäkra program och företagsägare som vill beskydda deras företags rykte.

Den centrala skillnaden mellan Monolitisk- och Mikrotjänstarkitektur och säkerhet kan igen spåras till skillnaden i kornighet. Mera specifikt innebär denna skillnad att Mikrotjänstarkitektur innehar säkerhetskännsliga kommunikationslinjer mellan de individuella mikrotjänsterna. Dessa kommunikationslinjer utgör ett område för attack som inte alls är närvarande i Monoliter. Detta leder till att individuella mikrotjänster kan bli kapade av cyberkriminella som kan använda den individuella mikrotjänstens rättigheter för diverse olika syften. Att säkra det ökade attackområdet i Mikrotjänstbaserade system utgör en stor utmaning. Först och främst är det svårt att bygga cybersäkra mikrotjänster. Dessutom är det svårt att övervaka och hitta fel i dessa p.g.a. systemets utspridda natur. Det finns också begränsad mängd information tillgänglig för hur man skall gå till väga.

Mikrotjänstarkitekturens finkorniga design medför dock även fördelar när det kommer till cybersäkerhet. T.ex. är det lättare att uppdatera de individuella mikrotjänsterna och därmed också snabbare att fixa sårbarheter i systemets cybersäkerhet. Monoliter som utvecklas under en lång tid brukar även samla på sig större och större mängder tredjepartsteknologier vilka kan vara svåra att byta ut då hela systemet hänger på dem. Ifall dessa tredjepartsteknologier visar sig ha säkerhetsluckor är det viktigt att snabbt byta ut dessa vilket kan vara svårt i Monoliter. Och andra sidan är detta inte alls lika svårt att göra inom individuella mikrotjänster vilka är mycket mindre med färre tredjepartsteknologier.

En annan aspekt är även Mikrotjänstbaserade systems förmåga att isolera individuella mikrotjänster. Vad som menas med detta är att ifall en individuell mikrotjänst identifieras som att kontrolleras av en inkräktare kan denna individuella mikrotjänst förstöras och ersättas av en ny duplicering mycket effektivt. Att göra samma med en Monolit är inte rimligt eftersom det skulle dra ner hela systemet och vara mycket långsammare. Detta är även kopplat till systemens förmåga att hantera fel i sina system. Om ett fel uppstår i en Monolit orsakar det ofta total kollaps av systemet. Ett Mikrotjänstbaserat system kan dock fortsätta att fungera i det stora hela även om individuella mikrotjänster kraschar.

Sammanfattningsvis kan det konstateras att de övergripande egenskaperna för Monolitisk- och Mikrotjänstarkitektur härstammar från kärnstrukturen hos respektive arkitektur. Mer specifikt ligger skillnaden i kornigheten mellan Monolitisk- och Mikrotjänstarkitektur till grund för de flesta av deras styrkor och svagheter. Mönstret till följd av skillnaden i kornighet visar att den finkorniga designen av Mikrotjänstarkitektur ger mer detaljerad kontroll över prestanda, skalbarhet, distribution och säkerhet. Samtidigt ökar finare granularitet komplexiteten genom att kräva mer detaljerade lösningar, vilka skapar nya utmaningar. Det kan sägas att möjligheten till kontroll kräver kontroll. Å andra sidan, på den andra sidan av spektrumet av kornighet, saknar Monolitisk arkitektur denna detaljerade kontroll, men uttrycker istället enkelhet. Det vill säga, mindre kontroll innebär mindre ansvar.

När det kommer till användningsområdet kan kornighetsmönstret användas som grund för argumentation. Monolitisk arkitektur upplever flera, till synes negativa aspekter i jämförelse med Mikrotjänstarkitektur som uttrycker förmågan till skräddarsydda lösningar på en individuell mikrotjänst basis. Samtidigt skall det inte försummas att Mikrotjänstarkitektur kräver nya lösningar som innebär nya utmaningar. Dessutom träder de flesta negativa aspekter av Monolitisk arkitektur fram först då programmet växer sig stort. Texten presenterar tre slutsatser om användningsområdet för Monolitisk- och Mikrotjänstarkitektur.

1. Så länge ett system är litet visar Monolitisk arkitektur sina styrkor, medan dess svagheter inte visar sig. Samtidigt medför Mikrotjänst utmaningar som kan betraktas som onödiga eftersom styrkorna med Mikrotjänstarkitektur först artar sig då ett system är större. Därmed lämpar sig Monolitisk arkitektur för mindre system och Mikrotjänstarkitektur för större system. 2. Om ett system kräver extrem skalbarhet är det möjligt att Monolitisk arkitektur inte räcker till. I så fall skall Mikrotjänstarkitektur användas. 3. De praktiska utmaningarna närvarande i Mikrotjänstarkitektur måste tas i beaktande. Om utvecklarna av en mjukvara inte är bekväma eller har tid att lära sig den nödvändiga teknologin, är Monolitisk arkitektur möjligen bättre.

Appendix B Usage of Large Language Models

In this thesis, I have used the large language model ChatGPT 3.5 for a couple of tasks. These are:

1. Before I had a clear picture of what I was going to write about I used ChatGPT to discuss various subjects. I found it was very good at brainstorming various ideas.

2. Once I had decided on writing about software architecture I used ChatGPT to find interesting subfields.

3. Once zoning in on a narrower subject I used ChatGPT to verify that the scope of my thesis was fitting for a bachelor's thesis in computer science. I found that it maybe wasn't the best at this task as, in my opinion, it often tries to stuff too much into the thesis. This was especially evident when asking it to provide a first draft of a table of contents based on a title I give it and some background information about what I want to be included and how long the thesis should be. Most of the time it overcomplicated things in my opinion.

4. When it comes to the actual writing of my thesis I used ChatGPT only to format BibTex sources, and put natural language into tables. These tables were only used for organizing thoughts and have not been used within this thesis. Lastly, I used ChatGPT a couple of times to formulate search strings for finding articles based on some instructions I gave it.

I have **NOT** used ChatGPT 3.5 (or any other AI Large Language Model) for information extraction or writing of text.