DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Implementation and Analysis of Authentication and Authorization Methods in a Microservice Architecture

A Comparison Between Microservice Security Design Patterns for Authentication and Authorization Flows

SIMON TRAN FLORÉN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Implementation and Analysis of Authentication and Authorization Methods in a Microservice Architecture

A Comparison Between Microservice Security Design Patterns for Authentication and Authorization Flows

SIMON TRAN FLORÉN

Master's Programme, Computer Science, 120 credits
Date: June 24, 2021

Supervisors: Mallu Goswami, Zeeshan Afzal
Examiner: Gerald Q. Maguire Jr.
School of Electrical Engineering and Computer Science
Host company: Ericsson
Swedish title: Implementation och Analys av Autentisering och Auktoriseringsmetoder i en Microservicearkitektur
Swedish subtitle: En Jämförelse Mellan Säkerhetsdesignmönster för Autentisering och Auktorisering i Microservices

Abstract

Microservices have emerged as an attractive alternative to more classical monolithic software application architectures. Microservices provides many benefits that help with code base comprehension, deployability, testability, and scalability. As the Information technology (IT) industry has grown ever larger, it makes sense for the technology giants to adopt the microservice architecture to make use of these benefits. However, with new software solutions come new security vulnerabilities, especially when the technology is new and vulnerabilities are yet to be fully mapped out. Authentication and authorization are the cornerstone of any application that has a multitude of users. However, due to the lack of studies of microservices, stemming from their relatively young age, there are no standardized design patterns for how authentication and authorization are best implemented in a microservice.

This thesis investigates an existing microservice in order to secure it by applying what is known as a security design pattern for authentication and authorization. Different security patterns were tested and compared on performance. The differing levels of security provided by these approaches assisted in identifying an acceptable security versus performance trade-off. Ultimately, the goal was to give the patterns greater validity as accepted security patterns within the area of microservice security. Another goal was to find such a security pattern suitable for the given microservice used in this project.

The results showed a correlation between increased security and longer response times. For the general case a security pattern which provided internal authentication and authorization but with some trust between services was suggested. If horizontal scaling was used the results showed that normal services proved to be the best target. Further, it was also revealed that for lower user counts the performance penalties were close to equal between the tested patterns. This meant that for the specific case where microservices sees lower amounts of traffic the recommended pattern was the one that implemented the maximum amount access control checks. In the case for the environment where the research were performed low amounts of traffic was seen and the recommended security pattern was therefore one that secured all services of the microservices.

Keywords

Authentication, Authorization, Access control, Microservices, Microservice Security, Security Tokens, Security Patterns, Performance

Sammanfattning

Mikrotjänster har framträtt som ett mer attraktivt alternativ än mer konventionella mjukvaruapplikationsarkitekturer såsom den monolitiska. Mikrotjänster erbjuder flera fördelar som underlättar med en helhetsförståelse för kodbasen, driftsättning, testbarhet, och skalbarhet. Då IT industrin har växt sig allt större, så är det rimligt att tech jättar inför mikrotjänstarkitekturen för att kunna utnyttja dessa fördelar. Nya mjukvarulösningar medför säkerhetsproblem, speciellt då tekniken är helt ny och inte har kartlagts ordentligt. Autentisering och auktorisering utgör grunden för applikationer som har ett flertal användare. Då mikrotjänster ej hunnit blivit utförligt täckt av undersökning, på grund av sin relativt unga ålder, så finns det ej några standardiserade designmönster för hur autentisering och auktorisering är implementerade till bästa effekt i en mikrotjänst.

Detta examensarbete undersöker en existerande mikrotjänst för att säkra den genom att applicera vad som är känt som ett säkerhetsdesignmönster för autentisering och auktorisering. Olika sådana mönster testades och jämfördes baserat på prestanda i olika bakgrunder. De varierade nivåerna av säkerhet från de olika angreppssätten som säkerhetsmönstrena erbjöd användes för att identifiera en acceptabel kompromiss mellan säkerhet mot prestanda. Målet är att i slutändan så kommer detta att ge mönstren en högre giltighet när det kommer till att bli accepterade som säkerhetsdesignmönster inom området av mikrotjänstsäkerhet. Ett annat mål var att hitta den bästa kandidaten bland dessa säkerhetsmönster för den givna mikrotjänsten som användes i projektet.

Resultaten visade på en korrelation mellan ökad säkerhet och längre responstider. För generella fall rekommenderas det säkerhetsmönster som implementerade intern autentisering och auktorisering men med en viss del tillit mellan tjänster. Om horisontell skalning användes visade resultaten att de normala tjänsterna var de bästa valet att lägga dessa resurser på. Fortsättningsvis visade resultaten även att för ett lägre antal användare så var den negativa effekten på prestandan nästan likvärdig mellan de olika mönstren. Detta innebar att det specifika fallet då mikrotjänster ser en lägre mängd trafik så är det rekommenderade säkerhetsmönstret det som implementerad flest åtkomstkontroller. I fallet för den miljö där undersökningen tog plats förekom det en lägre mängd trafik och därför rekommenderades det säkerhetsmönster som säkrade alla tjänster närvarande i mikrotjänsten.

Nyckelord

Autentisering, Auktorisering, Åtkomstkontroll, Mikrotjänster, Mikrotjänstsäkerhet, Säkerhetstokens, Säkerhetsdesignmönster, Prestanda, Belastningstestning

Acknowledgments

First and foremost I would like to thank the host company, Ericsson, for offering me the opportunity to make use of their resources when working on this thesis. I want to thank my supervisors, Mallu Goswami and Carl Rune Waite, who provided excellent support which assisted me in developing an understanding of their infrastructure.

From KTH, there are two persons in particular I would like to acknowledge. I want to thank my examiner, Prof. Gerald Q. Maguire Jr., for his eminent feedback which helped elevate the quality of my thesis. I also want to thank my supervisor, Zeeshan Afzal, for his invaluable support, especially regarding advice for the process of working on a master's thesis.

Lastly I want to thank my family and friends for their support. Even just being there helped me cope with the stress of working with the master's thesis during the COVID-19 pandemic we were all put through.

Stockholm, June 2021
Simon Tran Florén

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

API	Application Programming Interface
CWT	CBOR Web Token
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IoT	Internet of things
IPC	Inter Process Communication
IT	Information technology
JWT	JSON Web Token
LDAP	Lightweight directory access protocol
MSA	Microservice architecture
OASIS	Organization for the Advancement of Structured Information Standards
POC	proof-of-concept
REST	Representational State Transfer
RPC	Remote procedure call
SAML	Security Assertion Markup Language
SOA	Service oriented architecture
SSO	Single sign-on
URL	Uniform Resource Locator
vCPU	virtual CPU
XACML	eXtensible Access Control Markup Language

Chapter 1

Introduction

This first chapter presents in Section 1.1 a brief background of the area and describes why the project was performed, thus giving relevant context to the problem statement of Section 1.2. A research question is formulated based upon the problem that this thesis aims to solve. The purpose is explained in Section 1.3. Thereafter, goals and how they will be achieved through the selected research methodology are presented in Section 1.4. The delimitations are mentioned here to give a sense of what areas the thesis disregards in favour of focusing on the problem at hand. Finally Section 1.7 outlines the structure of the thesis.

The term *microservice* or *microservices* comes with some amount of ambiguity. It can refer to the individual components aimed to perform the business logic in cooperation with each other or the infrastructure as a whole. In this thesis, the latter definition is used, where a *microservice* is defined as the infrastructure containing multiple smaller *services* (among other important components) which perform their own unique business logic to achieve a specific goal but can also cooperate with other such services if necessary. The term Microservice architecture (MSA) refers to the specific architecture which *microservices* are built upon.

1.1 Background

As many developers working on large projects likely experience, handling large codebases comes with many difficulties. No one person is capable of being familiar with the entire code base, which can lead to difficulties in debugging or appropriate tests being hard to create. Deployment is another major issue since it requires an entire new instance of the application to go

online. In the worst case, the old application instance will have to be taken offline in order to deploy a newer one, leaving users unable to connect to services. Further development of projects that have grown large and become difficult to manage hampers scaling. For companies of a larger scale who need to maintain a competitive edge over competitors by growing their user base, this will eventually become unsustainable. Chris Richardson, author of the book *Microservices Patterns* describes this situation as dealing with a "Big Ball of Mud" and that the application may be in "monolithic hell" [1]. It can be interpreted as if all the different modules and components have grown out of proportion and created a web of dependencies so that it can no longer be comprehended or worked with in an efficient manner. The monolith is a possibility for an application's architecture but is described to be detrimental to further growth after a certain point. Therefore, another architecture is needed to address these issues while avoiding introducing more drawbacks than benefits. Enter the MSA.

As with any application, monolithic or otherwise, authorization and authentication are cornerstones of security. While the MSA presents possible optimizations for many aspects of the software development process, it also introduces new difficulties. Unsurprisingly, securing requests *to and from* but also *within* a microservice becomes a much more complex task when multiple parts of the application in a MSA need to communicate as compared to a monolithic application where the authentication and authorization can be done once when entering the application. The processes of authentication and authorization is seen as vital to create a secure MSA [2].

1.2 Problem

This section presents the problem and explains how it arises from an engineer's point of view. It is then distilled into the research question for the thesis project.

1.2.1 Original problem and definition

Since MSA splits an application into services, which are by the nature of the architecture meant to be **distinct** and **loosely coupled**, this makes the security of the whole application more complex to implement. The application, unlike a monolithic one, no longer has a single barrier of entry to the business logic. As internal communication between services is needed there are additional points where authentication and authorization may be needed depending on the security standards. If there is a large number of locations to secure, the

complexity of the security solution will rise. While MSAs are still possible to secure, there is no consensus about a widely adopted standard, partly due to the architecture being relatively new in software architecture terms.

1.2.2 Scientific and engineering issues

When implementing authentication and authorization in a monolithic architecture, a client only needs to communicate via one Application Programming Interface (API). In such an architecture, only the API receives requests to be subjected to authentication and authorization before the request is ultimately handled by the application. With MSA, the backend is split into multiple distinct services, all with different levels of security needs, which in turn has broadened the attack surface [3]. As the MSA concept is quite new, and is an *architecture* rather than a *framework*, no standardized method has been established for authentication and authorization. Moreover, the different suggestions from previous works in Chapter 2 have different levels of security. Some are satisfied with only securing the outermost layers of a microservice, while others claim that each internal request must be secured. Therefore, there is a trade-off between the simplicity of relying on a singular barrier to protect the MSA versus the complexity and possible additional overhead of multiple points of authorization within the MSA. Depending on how the services can be reached from outside clients, this may require either a single or multiple points to be secured.

While most of the existing literature looks at established authentication and authorization standards, security patterns are being overlooked [4]. Here security patterns refer to credible and predefined solutions applicable to problems within security (explained further in Section 2.4.1). Abbas Javan Jafari and Abbas Rasoolzadegan point out that these security patterns also lack practical research behind them [5] making further studies important. Understandably, established standards would be the first place to look when adapting new ways of securing a less understood area, such as MSA, but these older technologies were developed when other architectures were the norm. Therefore, it is also of interest to investigate how the previous methods for authentication and authorization could fit into new security patterns developed particularly for the MSA in order to increase security while maintaining the benefits provided by a MSA. It would then be necessary to quantify this research in order to prove its efficacy. Possible security patterns for authentication and authorization in a MSA could then define where and how the security checks take place. Anelis Pereira-Vale *et al.,* state that there is a

clear lack of security patterns for the MSA [4], so there is a need for research on this topic. A study of the current literature reveals that the different methods suggested for authentication and authorization have not been compared in a quantifiable manner under one and the same MSA which is another lack of coverage.

1.2.3 Research question

This then brings us to the research question addressed in this work:

> *What is the most appropriate security pattern to provide authentication and authorization in the given microservice and how does it compare to other possible patterns in terms of security and performance?*

The microservice in question refers to the one provided by the host company Ericsson.

1.3 Purpose

The purpose of this degree project is threefold. Firstly to develop a security solution which serves as a proof-of-concept (POC) that could in theory be applied to a full microservice. This can then assist in any future work on MSA security. Secondly as stated earlier, security in MSA is somewhat lacking in research and defined standards; thus, any project proposing a viable solution to this problem will add to the available research for future projects to build upon. Additionally, existing MSAs can draw inspiration and knowledge from such a POC. Lastly, this degree project course is a requirement for my graduation and this specific project was chosen to utilize what the author has learned for the duration his studies.

1.4 Goals

The goal of this project is to investigate how authentication and authorization can be structured in a microservice as a pattern to realize a security solution. Additionally, research must be conducted on existing attempts to secure microservices but also on general security technologies applicable to this scenario. When this is completed, the performance needs to be measured

to asses the impact of the added overhead between the implemented security patterns. The goal has been divided into the following two sub-goals:

Subgoal 1 Create a security solution which is tested and evaluated so that it may serve as the basis for further implementation.

The intention is that this research will not create a complete security solution but cover only a subset of the microservice which sees the most usage. This will serve as a POC which may be extended for full coverage of all services.

Subgoal 2 Present findings and evaluation in this thesis so that they may be reused or assist others in their further work.

As mentioned previously, there is a lack of research within the area of MSA, especially in regards to security. As far as the literature study was able to gather, no quantitative comparison between different authentication and authorization patterns was found. Therefore, the proposed research will be of value as a reference for further studies, both within Ericsson and the academic community. The ambition is that this will assist in legitimizing the methods used for authentication and authorization as security patterns in MSAs.

In addition to these goals of the thesis, a few deliverables for Ericsson were also to be fulfilled but are considered outside the scope of this thesis report itself. These were:

- Describe corporate security risks and threats. Research corporate vulnerability assessment strategies. And impacts of data breaches.
- Describe Ericsson's security for all the services running in the Kubernetes cluster.
- Ensure that all the existing and planned applications are covered.
- Understand possible solutions for Automated Audit Trail analysis and Intrusion Detection.
- Authorization and authentication support out-of-the-box to support security risk mitigation.

1.5 Research Methodology

The methodology consists of two distinct parts. First a literature study was performed in order to decide upon the technologies that are relevant for the project. It also helped identify and provide relevant background and knowledge needed to create a security solution inside a MSA, namely giving a fundamental understanding of MSAs and authentication and authorization. Related work was also identified to understand where the state-of-the-art was in terms of progress. This part is considered *qualitative*. Secondly, experiments were done to create POCs which could yield *quantitative* results useful in the evaluation of the proposed security solution. In order to acquire such data, load tests were performed. To evaluate the security, illegitimate attempts to access the microservice or certain parts of it were tested. All of these results were used in the evaluation of the POC and form a conclusion for the thesis.

1.6 Delimitations

MSAs can utilize many different services and databases. Since this project will consider different frameworks and technologies for use in a security solutions, there will be a need to both deploy and remove them when testing one framework after the other. Due to the limited time available for this degree project, this will be tested on a custom built set of services designed for fast deployment. This will limit the results applicability to the specific infrastructure but hopefully make the results more general.

The project will not evaluate the strength of the security solutions. It is assumed that the security employed delivers the defences claimed by the proprietor of the frameworks or standards used. However, the security was tested to see that no vulnerabilities were introduced as the result of poor integration of the technologies in the microservice used for the research.

1.7 Structure of the thesis

Chapter 2 covers the relevant information about microservices from a general perspective, common frameworks, technologies for authentication and authorization, security in microservices (including attack vectors), load testing for microservices, and finally related work. Chapter 3 provides a detailed description of how the microservice and the different security patterns were

implemented and deployed within the existing Kubernetes cluster. Chapter 4 describes the testing framework and specifications of the microservice as they pertain to the tests. Chapter 5 lists the results of the testing and provides an analysis of the recovered measurements. Chapter 6 discusses the results in general, how well the goals and the research question were addressed by the work performed, and it also makes use of the presented background and previous work to present the results in a broader perspective. Finally Chapter 7 makes some concluding thoughts, presents the limitations, proposes future work, and closes with some reflections tied to different aspects affected by the work performed in this thesis.

Chapter 2

Background

In this chapter necessary background information is provided to assist the reader in understanding the work done in this degree project. The chapter begins in Section 2.1 by describing what a MSA is, what it seeks to accomplish, the principles of the architecture, and why it is an important tool in creating efficient applications. Section 2.2 covers authentication and authorization methods. While there exist more ways than are mentioned, these were some of the most common technologies and standards. This gives some background regarding the state-of-the-art and gives an understanding of the technologies used in some of the POCs developed in this project. Section 2.3 will cover the Kubernetes framework and relate it to how it can be used in building a microservice. Section 2.4 will present methods on how to achieve a secure microservice and covers primarily security patterns, attack vectors, and how to integrate authentication and authorization technologies into a microservice. In order to grasp the full scope of state-of-the-art concerning MSA security, the literature study covers more than just the technologies used in the project work. Since these technologies were ultimately deemed to be unsuitable or of no importance to what was investigated; they are only presented in Section 2.5 as they represent contemporary research and alternative methods to achieve similar security goals. Section 2.6 presents some related work to showcase the inspirations and differences between what has been done previously and what this thesis project contributes, both in terms of the security solution implementation and creating custom testing frameworks. Finally, Section 2.7 concludes this chapter with a summary of the main points needed for the duration of this thesis report.

2.1 Microservice architecture

The idea of microservices has existed since the early 2000s but the term *microservices* started appearing in certain contexts around early 2010s [1, p. xviii]. However others claim it was coined as late as 2014 [6]. Regardless, the concept of microservices is quite young compared to other technologies and architectures within the area of software development. In order to understand where the MSA came from, it is important to understand how it relates to its predecessors, so that its distinct features can be identified. Olaf Zimmermann argues that MSA is a new approach to developing a Service oriented architecture (SOA), but with a stricter set of properties [6]. A SOA is an architecture focused on defining loosely coupled services with their own specific purpose, often realized through Web services and the use of Representational State Transfer (REST) [7]. However, one of the main focal points of the MSA is to divide the functionality into distinct services that do *not* directly depend on the rest of the infrastructure being in place. An attempt at defining the MSA in relation to SOA by Yarygina and Bagge found that there is no clear difference and that many of the same principles apply [8]. Despite this, there have been claims that applications built according to SOA are usually deployed as a monolith [9, 2], another architecture that stands in stark contrast to MSA. What is important to keep in mind is that MSA is not always superior to the monolithic architecture, but rather addresses some issues it may present in specific scenarios. Chris Richardson, has said that a monolith architecture actually has some benefits not found in MSA due to the ease of rapid development and growth of an application – as long as the application remains small scale [1, p. 4]. The benefits that come with the MSA are most effective when an application or service has grown to a substantial size. While the monolith architecture has tendencies to struggle with growth, an MSA, if done properly, excels at supporting continuous delivery and deployment, better code comprehension, and better maintainability, as the services are easily scaled and kept loosely coupled (explained in Section 2.1.1). Additionally, teams working on different features become more autonomous, enabling other languages and software technologies to be more easily utilized *without* disturbing unrelated parts of the application, and better fault isolation is provided [1, p. 14].

So what exactly is the MSA then? It is an architectural style that intends to split up a large code base that is handling business logic for all or almost all aspects of an application. While there are no strict set of rules and many aspects that can be done differently in a MSA, there are some design choices

that are consistent across descriptions of MSAs. The main feature of a MSA is to divide the business logic of the application into a number of distinct services. These services should not share a database but instead have their own, if the need arises. Furthermore, services can communicate through APIs with each other using Inter Process Communication (IPC) methods but can also communicate with external clients through either an API gateway or a similar proxy, or even directly (although this direct communication can lead to security issues as explained in Section 2.4.2).

In Figure 2.1, a possible implementation of a MSA is displayed where clients make requests through a gateway in order to make use of services. It is important to keep in mind that this is only a general picture of a microservices — since there is no widely recognized standardized architecture pattern. The API Gateway is not a requirement but it can be a useful tool when trying to merge accessible endpoints (possibly with different Uniform Resource Locators (URLs)) in to one singular API. A service can have the possibility to communicate with zero or more other services; however, the fewer services it can send requests to the better — since this decreases the attack surface as explained in Section 2.4.2.

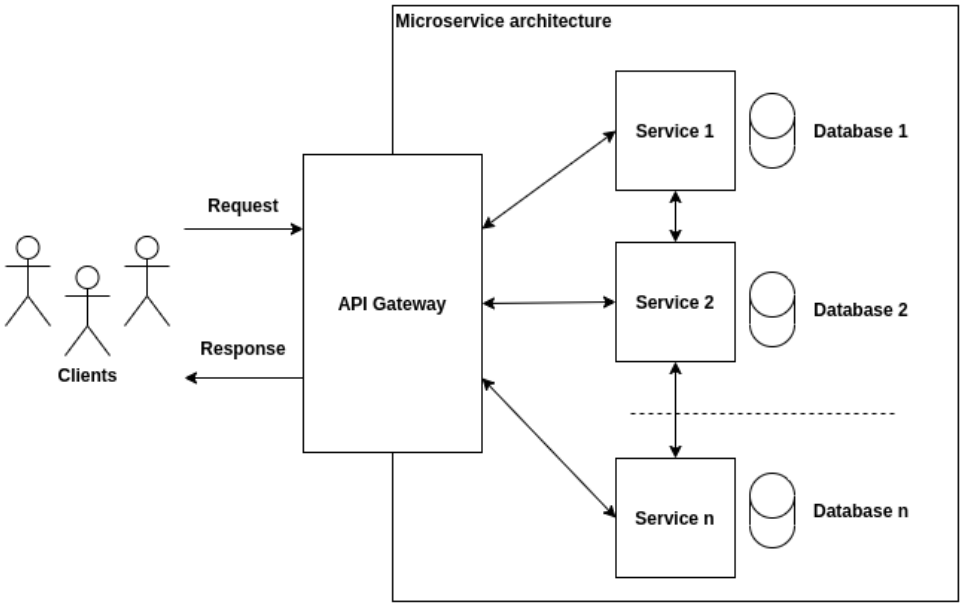

Figure 2.1: Graph displaying a microservice and possible communication flows.

2.1.1 Services

Services can be thought of as a subset of a backend with functionality related to a single business goal. This allows services to be programmed in the language best suited for their task(s) and gives developers a better understanding of the code — since the code that implements a service is decoupled from the rest of the microservice by an API layer [1, p. 15-17]. This facilitates testability and deployability by not requiring other features of the code base to work with the new changes [1, p. 15]. If the API both sends and delivers what is expected, then the rest of the application will behave as intended under the assumption that no other service or part of the MSA is faulty. What this means is that if something goes wrong in a service, *e.g.,* generating a stack backtrace, this fault will be isolated to the service at fault, making debugging easier. In addition, this error will not cause a failure in the entire system but rather only affect the service itself, making the infrastructure more fault tolerant [1, p. 16]. These benefits come from the loose coupling principle and is a pattern that should not be broken as it enables some of the core benefits of MSA.

In order to be able to have different services each with a potentially different programming language implementation, the deployment may become difficult. Fortunately there is software and tools specifically geared towards easier deployment. One could either make use of virtual machines [1, p. 390] or with containers [1, p. 393] which can easily be orchestrated with the use of Kubernetes* or an equivalent framework.

2.1.2 API Gateways

The API gateway acts as a proxy between services and any external client requests. Because this API gateway is a single choke point, it is a prime location for enforcing application security, such as authentication [1, p. 271]. Any requests that the API gateway receives, it will route to the appropriate servers. As with any API, this pattern promotes abstractions between clients and applications.

A concrete example is a network of Internet of things (IoT) devices which can be seen as services in a MSA and therefore have the architecture applied to it. An IoT device will then be protected by a layer of security implemented in a gateway around it, blocking unauthorized requests from the greater Internet [10]. The security can then be implemented using technologies such as OAuth2, JWT, etc., as explained further in Section 2.2.

* https://kubernetes.io/

2.1.3 Inter process communication

IPC is the method by which services and the API gateway communicate. One realization of such a method is Hypertext Transfer Protocol (HTTP) and by extension Hypertext Transfer Protocol Secure (HTTPS). Since HTTP can be used, this implies that it can be used in a RESTful manner [11]. Given the widespread use of HTTP on the internet, it has become quite popular for usage in IPC [12]. However, there are some drawbacks that exist, such as HTTP communicating *synchronously* which reduces availability [1, p. 103]. Since REST only has a limited number of verbs, it can be difficult to design a self-descriptive API. To solve this, one can turn to gRPC* which is a binary-based IPC. However, since languages like JavaScript are more tailored towards interfacing with REST or JSON-based APIs, it requires more work to use gRPC with these languages [1, p. 77]. This is reflected in REST being the popular choice as previously mentioned.

If one wants to achieve low latency, one should consider using an IPC with *asynchronous* messaging. This can be achieved with the help of a message broker [1, p. 86]. The message broker buffers messages in a queue and then delivers them as services are ready to accept new requests, thus improving performance.

2.2 Authentication and authorization

Authentication and authorization may sound similar but they are two distinct operations. Authentication occurs when a user wishes or needs to prove that they are who they claim to be. For example, when a user sends their credentials in the form of a username and password pair. If the system has a record of this unique pair of credentials, then the user is authenticated. In contrast, authorization is usually done when an authenticated user attempts to access some resource. Resources can be restricted such that they require a certain permission for an operation on this resource. Authorization checks that the user has the requisite permission. Therefore, when an authenticated user tries to access the resource, the authorization takes place and either approves or denies the access request.

Within the context of microservices, a common communication method is REST as it is widely adopted in other areas of IPC. Despite its popularity one should not assume that it is secure as it has been shown not to be inherently designed with security in mind [12]. Therefore, it requires additional practices

* An open source Remote procedure call (RPC) method introduced by Google.

and standards in order to remain a secure way of communicating. It has been claimed that the best suited method to secure REST is through token authentication and authorization, as explained in Section 2.2.2, but Tetiana Yarygina has said that this is not as secure as methods using keys to sign requests [12]. Still token authentication and authorization provides a feasible trade off between security and scalability.

Authentication and authorization are among the most common and effective means for realizing application security. Therefore, standards and technologies have been developed to realize good authentication and authorization. The following subsections describe some of the most used technologies and standards to enforce this authentication and authorization.

2.2.1 Basic access authentication

Basic access authentication is a scheme that enables HTTP requests to use the header `Authorization` in order to provide a Base64 encoded string which a server can validate. When a protected resource receives a request without the appropriate credentials in the header, a challenge to authenticate is provided in a response with code `401` and header `WWW-Authenticate` is returned [13].

As the scheme name implies, this is only a basic way to achieve authentication and authorization. In the following subsections, standards that are common in industry are presented.

2.2.2 JSON Web Token

The HTTP protocol is stateless which means that there is no built-in way for a server to remember any communication with a client. In order to protect resources there is a need to remember a previously authenticated and authorized client in subsequent HTTP requests lest they need to be reauthenticated and reauthorized. In order to maintain this state in which a user is verified, a token can be sent with every subsequent request containing user information and permissions. A standard for this is the JSON Web Token (JWT). It can be used for transmitting said information in the form of a JSON object, which in turn can be either signed or encrypted to provide integrity or secrecy respectively [14]. In the article by Rongxu Xu, Wenquan Jin, and Dohyeun Kim [15] they make a proposal as to how a MSA can be secured through the usage of JWT. In this method it is assumed that an API Gateway intercepts all requests so that an authorization server can provide JWTs for subsequent requests to services handling sensitive data.

2.2.3 OAuth 2.0

OAuth is an open standard which decreases the number of authorization steps by asking a user to allow a service authorization to other services containing private data [16]. Today, the OAuth protocol is considered obsolete as its next version fulfills the same purpose but has few implementation similarities [17]. The new version called OAuth 2.0, simplified as OAuth2, is a protocol which can be used to define the information flow when performing authorization (see Figure 2.2). OAuth2 is an industry standard and is managed by the Internet Engineering Task Force (IETF) OAuth Working Group*. As shown in the figure, the flow defines four roles: a resource owner capable of issuing grants for authorization, the resource server which needs access tokens before serving a request, the client who creates the initial request for a protected resource, and the authorization server responsible for authentication and issuing access tokens [18]. One method for the authorization server to lookup the permissions of a given user is through Lightweight directory access protocol (LDAP). Despite not being specifically mentioned by the OAuth2 flow, LDAP is still a popular approach [19]. With OAuth2, authorization can be delegated to a remote server which allows a user to access other protected services *without* having to manage multiple accounts. While not necessary, OAuth can use JWTs as its issued access tokens; thus, taking advantage of JWT's sessionless properties [20]. In terms of how this can work with a MSA, one proposed solution has clients communicate with an OAuth2 server (which lies outside the microservice) and receives a JWT it can use to access the services without any further need for authorization [21]. If one wishes to use OAuth2 for authentication, OpenID Connect† exists as an extension to allow for this functionality [22].

2.3 Kubernetes

OS-level virtualization has become extremely popular with Docker ‡ being a significant name behind the technology. It boasts its usage by large companies§ which points to its popularity. With increased usage, it is common for frameworks and tools to be developed with the intent of enhancing functionality. For Docker one common container orchestration tool is Kubernetes¶. It allows easier management, deployment, and scaling of Docker

* https://datatracker.ietf.org/wg/oauth/documents/
† https://openid.net/
‡ https://www.docker.com/
§ https://www.docker.com/customers
¶ https://kubernetes.io/

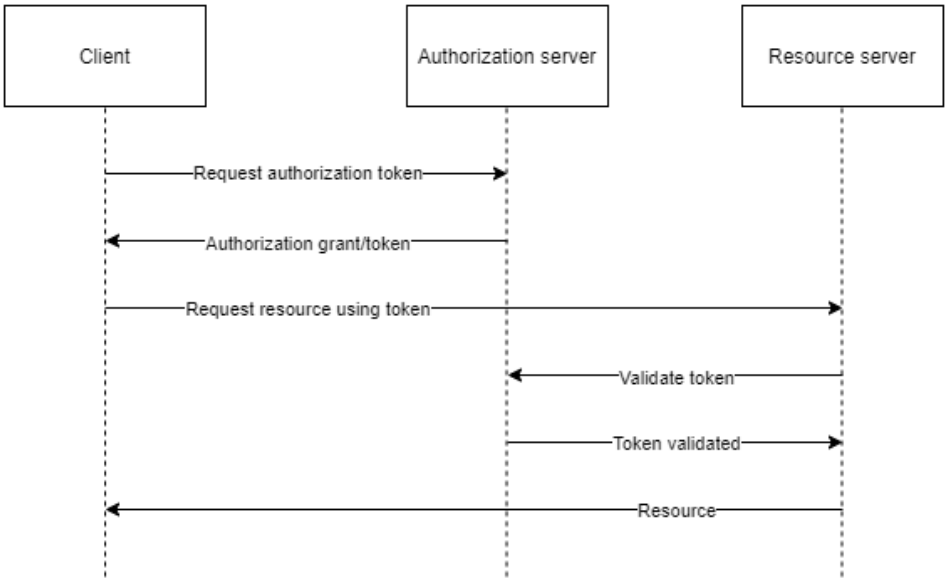

Figure 2.2: Simple overview of the information flow in OAuth2

containers. Kubernetes can handle different containerized applications which are deployed in a cluster where they can communicate with one another or endpoints can be set up which allow requests by clients outside of the cluster. This structure has many parallels with the MSA which makes it an attractive environment to host a microservice.

2.3.1 The cluster

When one makes use of Kubernetes it is usually done by setting up a cluster in a cloud environment. From there it can be further divided up into namespaces. In order to run an actual application, the application must first be containerized by providing a Docker image of the application. What is known as nodes can then run the applications in pods (explained further in the Section 2.3.2) as if they were running on a normal server. The nodes are controlled by Kubernetes to allow for automation of certain tasks, such as scaling and management.

2.3.2 Pods

The actual application is hosted as pods running in nodes. In order to scale horizontally more pods can be added to have multiple instances running.

Kubernetes handles any logistical issues and abstracts this to the user as a choice of the number of pods that they want to have running.

2.3.3 Services and ingress

All pods running in the cluster are unreachable from the outside by default. Nor can the different applications running as pods communicate with one another unless their internal cluster IP address is known. Since this cluster IP address is not predefined, this poses an issue if there is a need for cooperation between the different applications. In order to allow reliable internal communication, a service can be created to grant the pods in a node a domain name. Still this is not enough as this DNS name will only work inside the cluster. Hence the use of what is called ingress. An ingress object defines how a service can be reached by outside traffic by giving it a URL that can be accessed through a DNS server either publicly or in a larger private network that the cluster is part of.

2.4 Security in microservices

As mentioned in Section 2.1, the concept of MSA is quite young, hence there has been only limited research, with security being one specific topic that is largely neglected. This is revealed by systematic surveys of research regarding security in MSAs. These surveys show that authentication and authorization are the leading methods used to address security issues and the majority of prior research has focused on these methods [4, 23]. However, the same surveys remark that there is a lack of security pattern research and proposals of how to detect and mitigate attacks. A better definition of security patterns is given in Section 2.4.1.

The current state-of-the-art regarding authentication and authorization in MSA often use established technologies to build their own solutions. A model proposed for authentication and authorization usage within healthcare systems weighed different benefits and drawbacks of different security solutions for authentication and authorization and ultimately selected the industry standards OAuth2, Single sign-on (SSO), JWT, and OpenID [24]. However, the majority of proposed solutions and POCs found in the literature, do not feature all four of these industry standards but rather a subset of them. This may be because while they can be used in conjunction they offer very similar functionality in terms of authentication and authorization possibilities. An example is SSO and OpenID which both provide authentication but realize it through different

implementations. Some of the most common approaches use either an API Gateway to provide JWTs [10, 15, 25, 26] or the OAuth2 flow [27, 28].

2.4.1 Security patterns in microservices

Security design patterns, or more simply security patterns, are derived from software patterns [29] which in turn come from patterns used in architecture and design [29][1, p. 20]. Software patterns aim to encompass expertise in a standard so that it may be applied to the relevant project or problem in order to guide the design process during development [29]. Security patterns apply the same concept to the realm of software security [29]. In a MSA, different security patterns describe where authorization could be enforced, either at a single point or at multiple points throughout the microservice. Another pattern concerns how this authorization is implemented, *e.g.,* as a disconnected proxy to the services handling sensitive data or directly in the code for the service. These alternatives provide different benefits in terms of complexity versus security. Securing only the edge level API Gateway (as in the case of Figure 2.1) can be seen as the minimal amount of security; however, it supports the principle of loose coupling since it allows developers to work independently on services without having to focus on security. One should still be wary when considering practicality over security since this may lead to a *confused deputy attack* in which a compromised service has full access to the rest of the services due to there being complete trust of any requests made from within any service inside the microservice [2]. Since services need to expose an API for a gateway or other services to communicate through, there is an increased attack surface. Therefore, another pattern where each service is independently secured will provide more fine-grained authorization security and enforce the concept of defence-in-depth, explained in more detail later in this subsection. A possible setup of this is shown in Figure 2.3. However, this diminishes one of the main benefits of microservices which is to allow the developers of a select service to **only** focus on the core functionality that it should provide. To simplify the aforementioned pattern, gateways can be setup for each service to remove the security aspect from the service's code base [28]. As one can imagine, this may lead to an excessive number of API Gateways and adds an extra step of having to go through each gateway for any request irregardless of whether the request comes from a client or a service within the microservice itself — as shown in Figure 2.4. As was demonstrated in [28], a compromising alternative assigns one gateway to multiple services that need the same level of security. This is visualized in Figure 2.5 where

additional layers of security are provided by secondary internal gateways but all communication within the microservice is assumed to be trusted. In all three examples, having the authorization as its own autonomous service within the MSA will provide the benefits associated with loose coupling [2]. While all of these ways of providing authentication and/or authorization could be seen as security patterns, they have not officially been recognized as such [4].

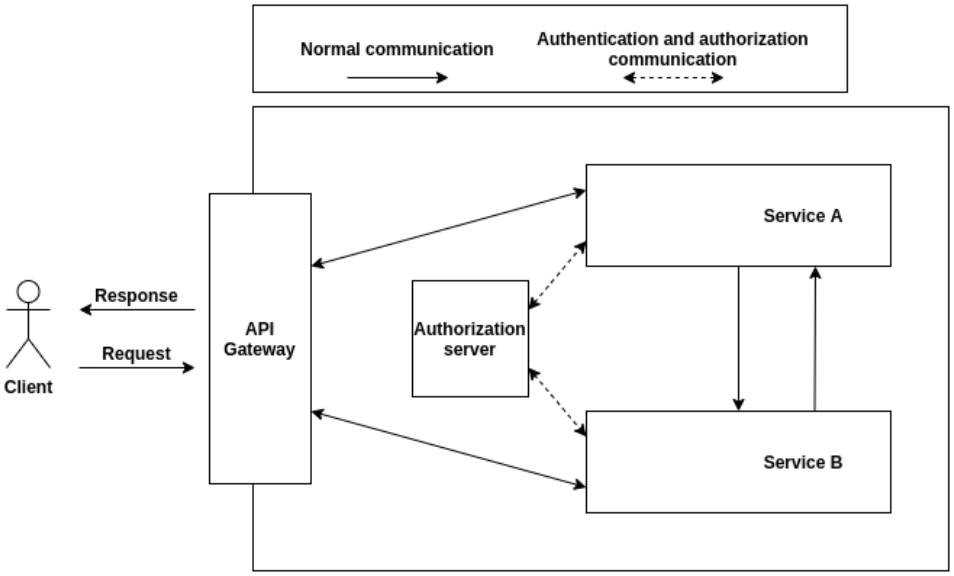

Figure 2.3: Graph displaying communication flow when using internal authorization in services

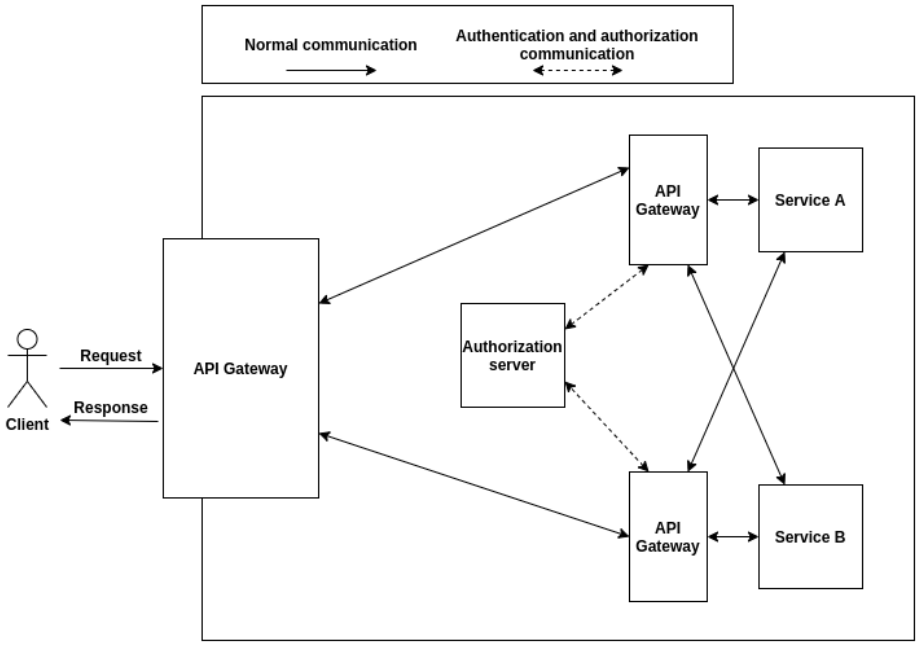

Figure 2.4: Graph displaying communication flow when protecting each separate service with its own API Gateway

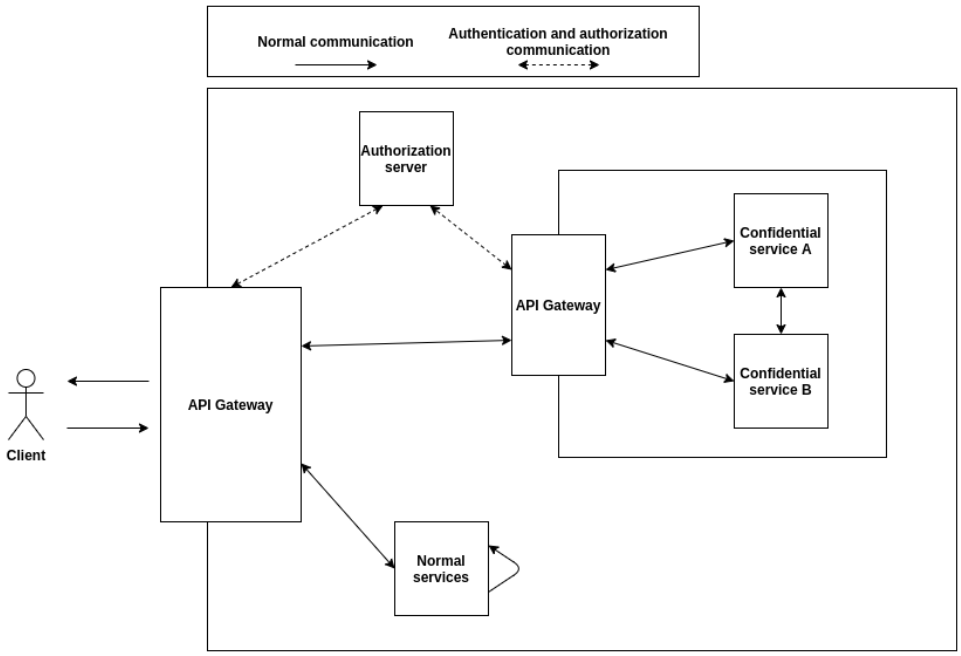

Figure 2.5: Graph displaying communication flow when a subset of services requiring extended security measures is given its own API Gateway

Defence-in-depth is the concept of setting up defences at multiple layers in order to prevent attacks getting through the outer perimeter as a means of setting up contingencies. In microservices, defence-in-depth would set up multiple gateways as extra layers of security beyond the outermost API Gateway or have each of the services handle authorization themselves. As additional security, separate encryption keys can be used to secure service communication thus preventing attackers who have gained control of a service from being able to access other services handling more sensitive data [30].

2.4.2 Attack vectors

As explained by Otterstad and Yarygina [31], the main attack vector to always be concerned about is any code accessible to an attacker. Hence there is a need to assume malicious intent for all requests sent by any user since these requests need to be processed by a service. This leads to the importance of the API Gateway. If services are not secured by a unified entry point, then the attack surface increases many fold (at least proportional to the number of entry points). The risk is that an instance of a service could be controlled by a malicious party. Otterstad and Yarygina go on to show that this initial attack vector can lead to further exploits. By escaping to a hypervisor orchestrating the services, possibly as containers, an attack can reach other services hosted by the same hypervisor. Even if there is a separate hypervisor responsible for the other services targeted by the attacker, there is still a need to perform an initial exploit towards the parts of the services normally reachable from the outside. A takeaway from this is the importance of the initial security between a client and the MSA.

To look at security beyond the barrier provided by the API Gateway located at the edge of a microservice, even the services themselves may need to be considered as potential attack surfaces [32]. Here it needs to be assumed that every possible communication, even from the services themselves, cannot be trusted which leads to a need for a multitude of security checkpoints that encapsulate each of the services [8]. As one can assume this is more costly in terms of resources and may tax the system, thereby presenting a dilemma necessitating a trade off between security and productivity as pointed out by Daniel Richter, Tim Neumann, and Andreas Polze in [33]. Their paper goes on to mention that if security is an afterthought it may become very difficult to identify clear benefits to motivate better security; therefore, planning in security from the very start of creating a microservice, possibly with the help of security patterns, can be instrumental in reducing the attack surface.

There are scenarios where measures against outside attackers might not be enough. Within corporations there is always the risk of internal attacks. These attacks can come from, but are not limited to, corporate espionage or disgruntled employees. In a study to generate a framework against such attacks, Kai Jander, Lars Braubach, and Alexander Pokahr showed that since services of a MSA are suitable for deployment in containers (*e.g.,* Docker containers in a Kubernetes environment), internal attackers could compromise the containers and then cause damage or leak data from the system [34]. Such an example points to the importance of roles and the principle of least privilege which from the point of view of microservices security translates to services not having contact with services that they do not require cooperation with [31].

2.5 Other MSA security measures: outside the scope of this project

This section describes a set of important methods to achieve a secure microservice. These methods offer different benefits in authentication and authorization as well as other areas of security (such as attack detection and recovery). However, they were either insufficiently related with the goals and purpose of this thesis project or had a scope which would take far more time than was available to perform in the scope of this thesis. Despite this, they are summarized in this section to give a more complete overview of what the latest research and state-of-the-art is and also to give the reasoning why they were excluded.

2.5.1 Intrusion detection and response in microservices

According to Anelis Pereira-Vale, *et al.,* much of the prior research lacks a focus on "reacting to" or "recovering from" [4]. To address this, Tetiana Yarygina and Christian Otterstad applied game theory to a intrusion response system that is able to monitor a microservice and make decisions that will minimax the defensive outcome [35].

Nehme *et al.,* suggest in their article that an API gateway, assumed to be located at the edge of the system, is a possible location to setup an intrusion detection system [2]. However, they also mention that services will need their own custom signature which can be difficult to implement. Further, Sun *et al.,* mentions in their article that this approach is not effective against internal

threats [36].

In this thesis project, this approach to security was not pursued. Although, it is in the realm of necessary research for MSA security, it attempts to provide a different type of security and is too distant from the goals and purpose mentioned in Sections 1.4 and 1.3 (respectively). However, it was researched for possible future work that is of interest to Ericsson.

2.5.2 Using encryption keys in authentication and authorization

While most of the literature uses industry standards when it comes to authentication and authorization, such as JWT, OAuth2, or SSO to only name a few, there have been attempts at using encryption keys to provide these security aspects. Kai Jander, Lars Braubach, and Alexander Pokahr used this method to secure the internal calls between services, with a key exchange performed between two communicating parties where roles are associated with a key in order to achieve the access control [30]. Encryption keys were considered too dissimilar from the technologies selected for this thesis project; hence, they were not used.

2.5.3 SAML and SSO

While OAuth2 and JWT are widely used for authorization*, SSO assists in decreasing the number of login actions required and focuses mainly on authentication. SSO uses a store of identities related to different services (not necessarily MSA services) as a sort of key chain for login credentials, all accessible with one single credential [38]. Whereas OAuth2 is a standard for using access grants, possibly in the form of JWT; Organization for the Advancement of Structured Information Standards (OASIS) Security Assertion Markup Language (SAML) 2.0 is a standard for how to use SSO. SAML 2.0 works in a similar fashion to OAuth2 but instead of requesting authorization to a resource, it requests to be authorized to access a service based on a single login. A client first sends their credentials to an **identity provider** from which it gets the actual authentication details for the service it wants to access [39].

While SSO and SAML are closely related to authentication, the literature study determined that rather than focusing on delivering security, SSO and SAML were more focused on providing convenience. The argument is then

* even though JWT is suitable for authentication as well [37])

that in order to more easily establish the relevant factors that explain the results of the project, irrelevant variables should be removed; hence, SSO and SAML were not included in this thesis project.

2.6 Related work

This section summarizes research related to the work done in this project. This covers using open standards and common technologies, such as OAuth2 and JWT, in a microservice. It also presents what could be seen as suggestions for different security patterns in MSAs. In Section 2.6.4, IoT is mentioned since a fair amount of research has gone towards securing IoT devices using techniques related to security in microservices where different standards and security patterns are considered.

2.6.1 Microservice load testing and security design patterns

While this thesis report focuses on performance impact of security design patterns, it can still be of interest to draw inspiration from how a performance analysis is done with other aspects in mind. One such example is a comparison between the MSA and serverless computing conducted by Chen-Fu Fan, *et al*. Their research set up one instance of both architectures and sent varying number of requests to measure response times [40].

Another related work is in an article by Akhan Akbulut and Harry G. Perros [41], comparing design patterns for microservices (not security patterns). Much like the work done in this thesis, one of the patterns implemented is tested by simulating different levels of load by having threads act as users sending requests to the microservice constructed as a test scenario. For the remaining two design patterns, the tests measured other metrics, such as resource usage. As the different patterns are quite dissimilar there is not much focus on direct comparison but there is still an emphasis on showing strengths and weaknesses for using the patterns in different environments.

2.6.2 Architecture patterns for authentication and authorization

Because of the loosely coupled nature of microservices, there is an added complexity in securing them. This stems from the fact that instead of securing

a single application there are now multiple smaller applications with varying amounts of critical data. Therefore, there may be many more locations where enforcement of authentication and authorization must or should exist. Most previous research focuses either on only securing a microservice on the API Gateway level as in [25, 10] or goes for a more defence-in-depth approach [30, 28]. However, no research provides a quantitative comparison between different patterns when used in a practical setting.

2.6.3 Standardized technologies applied to microservices

A majority of research papers have utilized some existing standard or framework in order to perform either authentication or authorization. It makes sense that the methods accepted by both organizations, such as IETF, and companies, would be prime candidates to begin the task of securing a microservice.

The simplest way of authenticating a user and providing access control is to do so for each incoming request. However, this can lead to poor performance if the overhead per request is high and the rate of incoming requests is too high. Olga Safaryan, *et al.,* proposed a solution to secure a microservice using Zuul* together with JWT in order to bypass the need for any subsequent authentication [25]. W. Jin, *et al.,* proposed a similar concept that also uses JWT but where all security services were placed in the gateway [10].

Some research has been dedicated to defence-in-depth which goes beyond the single layer of security that API Gateway authorization provides. Kai Jander, Lars Braubach, and Alexander Pokahr [30] proposed a protocol for encrypted message exchange in a microservice. However, they mention that the use of JWT is too complex to be deployed if one does not have the necessary background knowledge. Antonio Nehme, *et al.,* [28], investigated one of these patterns, specifically to have each service protected by its own API Gateway where authorization is performed with tokens issued according to the OAuth2 flow. This helped guide the practical work done in this thesis when it came to what directions to take with the patterns more focused on defence-in-depth. Another approach to simplify this pattern is to have a single authorization server implemented as a service in the MSA. This was also suggested by Xiuyu He and Xudong Yang in [26] and provided some background which assisted in the selection of pattern implementations.

* https://github.com/Netflix/zuul, https://netflixtechblog.com/open-sourcing-zuul-2-8

As the microservices used in this thesis project are developed using Spring Boot*, some inspiration were drawn from previous work which focused on using the Spring Boot framework together with OAuth2 in order to secure a microservice [27]. However, this earlier study makes no mention of JWT usage and it does not measure the overhead imposed by the security solution nor does it compare any security patterns. This motivated the examination of JWT usage and measurement of its overhead while comparing multiple security patterns within the current thesis project.

2.6.4 Security needs when applying MSA to IoT

There have been some positive results in applying MSA's principles and architecture to similar systems. The distributed nature of IoT lends itself well to MSA since it also deals with distinct services that need to communicate with one another. The devices themselves can be seen as services, working towards their own unique business goals and with their own set of data similar to a service's database [42]. As expected, the solution that MSA offers also brings along problems and security is no exception. In order to combat this, eXtensible Access Control Markup Language (XACML)† policies were proposed to be used to secure the network from the greater Internet [43]. This is indeed very similar to securing a microservice at the gateway level. A full on MSA is possible to integrate with IoT devices as shown in [15], where the security was realized through JWT and authentication and authorization were performed in the Kong API Gateway [44]. This made the architecture almost indistinguishable (in terms of security) from a MSA created for an application.

Another angle to view the MSA in with IoT in mind is to see the devices as clients. Dimitrios Kallergis *et al.,* [45] proposed CAPODAZ, an architecture to utilize microservices, in order to promote scalability and stability of resource servers used by the IoT devices. This architecture was created with authorization in mind and used CBOR Web Token (CWT) [46], a binary version of JWT, to provide a natural way for simple devices to be authenticated.

2.7 Summary

This chapter has covered the essentials of MSA along with a summary of the state-of-the-art options for authentication and authorization. Additionally,

* https://spring.io/projects/spring-boot
† https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

there was a summary of research into security specialised for microservices, with a focus on authentication and authorization. While the methods are important to understand, it is also of great benefit to understand why security measures are needed and what they protect against. Hence, attack vectors were explored. All this information culminates into patterns that aim to be reusable strategies when a microservice is built. Based on the literature study, the following security patterns for authentication and authorization were identified with the associated benefits and drawbacks presented in Table 2.1.

Table 2.1: MSA authentication and authorization security patterns

Pattern	**Benefits**	**Drawbacks**
Edge level gateway	Simplifies architecture and minimizes communication per request	Single line of defence
Service group gateway	Services with the same security level needs will be secured behind the same gateway. Others can be left as is or have their own gateway	Requires initial and concurrent knowledge as to what services needs to be secured when creating the infrastructure and building upon it, respectively
Service level gateway	Each service is secured individually, mitigates internal attacks	Worse scaling since each new service adds another gateway, more communication needed per request

Chapter 3

Authentication and authorization in a Kubernetes microservice

This chapter goes into the technical details of the components implementation required to setup authentication and authorization. The components necessary were the authentication and authorization service (shortened to auth-service) coupled with a Redis store, gateways, and some sample services (which emulates the business logic of a microservice). This was all done in a Kubernetes cluster which provided both benefits and challenges. Section 3.1 covers the research process including the literature study and how it informed the implementation of the microservice and the developed patterns. Section 3.2 discusses the Kubernetes cluster where the microservice was deployed and how communication to and from it is possible. Section 3.3 introduces the service in charge of handling login and the corresponding sessions as well as the authorization and authentication. In Section 3.4, the gateway implementation is presented. Next, Section 3.5 describes the services which perform what would be the business logic of the microservice. Finally, Section 3.6 presents the different security patterns that were studied and shows their architecture.

3.1 Research process for microservice and security pattern development

The project was initiated by performing a literature study. The findings are part of what was presented in Chapter 2 which serves as the background of the thesis. While some of the information that Chapter 2 is to give a broader view of the areas that apply to the project work, it is also responsible for directly influencing the choices made when creating the microservice and the security patterns. Both general information regarding MSA (found in Section 2.1) and specific details about Kubernetes (found in Section 2.3) was used to design the microservice within the host company's cluster. Much of the related work that was presented in Section 2.4.1 served as the inspiration for the security patterns. Both recurring and unique concepts were used when deciding on security pattern design which can be viewed in Section 3.6.

3.2 The Kubernetes cluster and ingress

The microservice is realized using Kubernetes. Since all traffic inside the cluster is unreachable from outside, what is known as an ingress controller facilitates communication originating from outside (*i.e.,* the internet or the local network that the cluster is deployed in) into the cluster. The Kubernetes cluster that the microservice is deployed in uses an NGINX ingress controller variant [*]. The service that one wishes to expose (*e.g.,* an edge level gateway as in this implementation) will be assigned an ingress object (which defines how the ingress controller should route traffic related to the service) which designates a reachable URL if the requests comes from a device connected to the internet or an internal network. In this setup, the cluster is only reachable from an internal Ericsson network.

3.3 Auth-service and the Redis store

The auth-service is structurally similar to an Express.js[†] backend. Endpoints are used for the other services to communicate when the need arises, most notably a request requiring authorization will be rerouted to one of the endpoints with such capabilities. The auth-service can login users by

[*] https://github.com/kubernetes/ingress-nginx/blob/master/README.md
[†] https://expressjs.com/

communicating with an LDAP server which also responds with the roles of a user. By giving a logged in user a JWT, this token can easily be used to both identify a user for authentication and also to find the associated roles of this user for authorization.

For session storage, Redis* is deployed in the cluster. Redis is deployed in accordance to Ericsson's standards and is only accessible via the auth-service. This way after a user has logged in, information, such as roles, are readily accessible by the auth-service alone without further need to access the LDAP.

3.4 Gateways

The gateways used in the different security design patterns for authentication and authorization utilize Express.js much like the auth-service mentioned in Section 3.3. They are lightweight in terms of code implementation and makes use two endpoints in the auth-service to either authenticate incoming requests or authorize them. If either of these security checks are used and return a status code of 200 the request will be passed on to the referenced service of the initial incoming request. If at any point a response should have another status code (401 if the request is unauthorized or cannot be authenticated) the gateway will itself return the very same status code to inform that the original request could not be sent past this point. The gateways are designed so that they all have identical source code. This enables one independent team to maintain and develop the gateways irregardless of whether the pattern has a single gateway at the edge of the microservice or a gateway at each service.

3.5 Services

The services are written in Spring Boot† and provide simple REST endpoints which can either directly return some value or send another request within the microservice to get additional resources. As the services were meant to be consistent and generic they were given an artificial work delay of 200 ms. This is in order to mimic the behaviour of a microservice in a real setting but without any unforeseen delays taking place. All other work performed by the services was also identical. Since the response time when the service is meant to perform specific tasks the time it needs depends wholly on what function is being performed. Hence a delay which is capable of ensuring that

* https://redis.io/ † https://spring.io/projects/spring-boot

threads are not able to be created and have their response sent back faster than the load testing software can create them was chosen. This means that the response times themselves should not be used as an indicator as to how well a security pattern performs and should instead be used when comparing relative performance. What is also important to note is that the delay does not consume CPU resources as arbitrary calculations would which means that resource starvation is not what slows the system down but rather congestion from requests being put in a service's queue.

To let a service know if it should contact another service or directly return its value a string containing a sequence of services to be contacted can be sent in a request. Each time such a request is received by a service it removes itself from the sequence and looks at the next location to send the request. Each service has an associated role that the user is required to possess for the auth-service to allow access to the service.

3.6 Authentication and authorization patterns

This section goes into detail of how the three different security patterns for authentication and authorization were implemented: edge level (§3.6.1), service group (§3.6.2), and Service level (§3.6.3) gateway patterns. In Section 2.4.1 figure 2.3 the literature study details another pattern where the services themselves are developed to have the functionality of what the gateways described in Section 3.4. As this was considered going against the MSA principle of services being solely focused on their unique business goal it was not implemented. The implementations using these security patterns presented here is what would become the different POCs for the security solutions.

3.6.1 Edge level gateway pattern

The edge level gateway pattern is the simplest of the three security patterns. It was identified as the most common way of implementing authentication and authorization in a microservice but it provides the minimum amount of security. Therefore, it can also be seen as a baseline to compare the other two patterns to (as both are more complex and provides a higher level of security). All services present in the microservice are unreachable by outside clients or servers without first sending a request to the edge API gateway. This is

achieved by only exposing this Kubernetes node using ingress, as covered in Section 3.2. When a request is received it checks for a JWT which if present is passed on to the auth-service for authorization. Authentication is done whenever the issuer of the request claims their JWT. If the token is present and verified by the auth-service, then the gateway routes the request to the indicated service. If the token is not present or is invalid, then the gateway simply sends a response with HTTP status code 401, *i.e.,* Unauthorized. The only place that a gateway is deployed is the edge API gateway; hence, the name 'Edge level gateway" pattern. All communication between services behind the edge gateway is allowed to be preformed directly without further token validation. A visualization of this can be found in Figure 3.1.

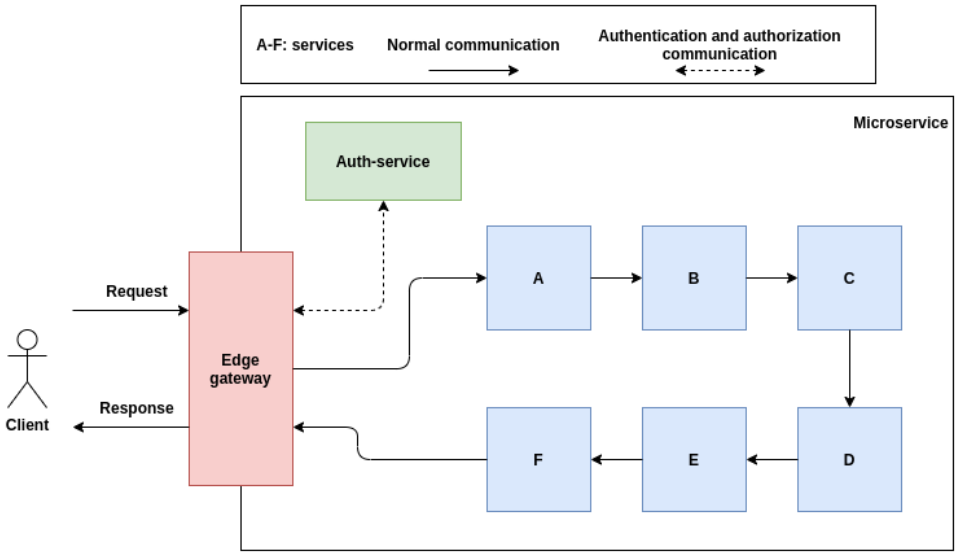

Figure 3.1: Visualization of the edge level gateway pattern implementation in the microservice

3.6.2 Service group gateway pattern

The service group gateway pattern builds upon the edge level gateway pattern but groups together services which requires the same level of access to reach. This authorization is also handled by the auth-service. An example would be a set of services that all requires the same role to access to access them. This subset of services would much like the edge gateway protects all existing services be protected by an additional gateway which itself is also behind the

edge gateway. This creates an additional layer of security. Another similarity to the edge level gateway pattern is that communication that does not require to pass through a gateway is not subject to authentication or authorization. This means that services behind the same internal gateway are free to send requests without the need for authentication and authorization. A visual representation of how this pattern is implemented can be seen in Figure 3.2.

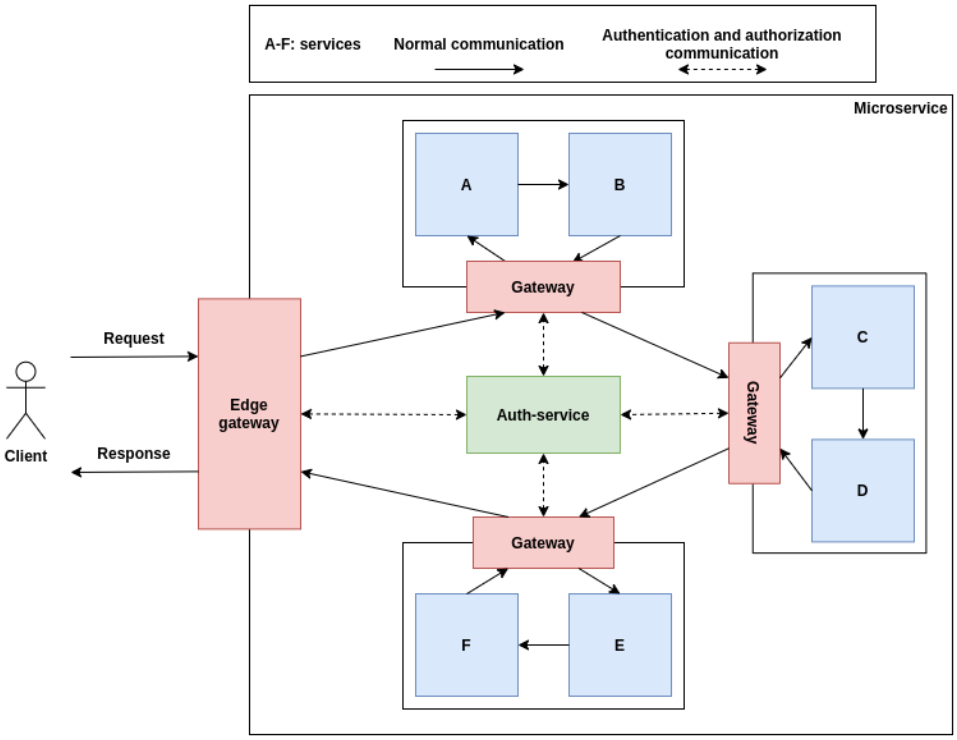

Figure 3.2: Visualization of the service group gateway pattern implementation in the microservice

3.6.3 Service level gateway pattern

The third and final pattern considered in this thesis is the service level gateway pattern. This security pattern requires every service to have its own gateway which protects it through authentication and authorization. As it may not be feasible in all situations for there to be one role associated with one service, some or all of the gateways can have the same role required to allow access to the protected service. While it might seem unnecessary to perform role verification when two service-gateway pairs communicate and require the

same role to access, this actually provides a unique form of protection. As all communications within the microservice between its services needs a security check, services controlled by a malicious actor has no other possible targets accessible without first going through authentication and authorization. In the edge level gateway pattern all services would become vulnerable and in the service group gateway pattern services behind the same internal gateway would be affected. The corresponding implementation visualisation can be viewed in Figure 3.3.

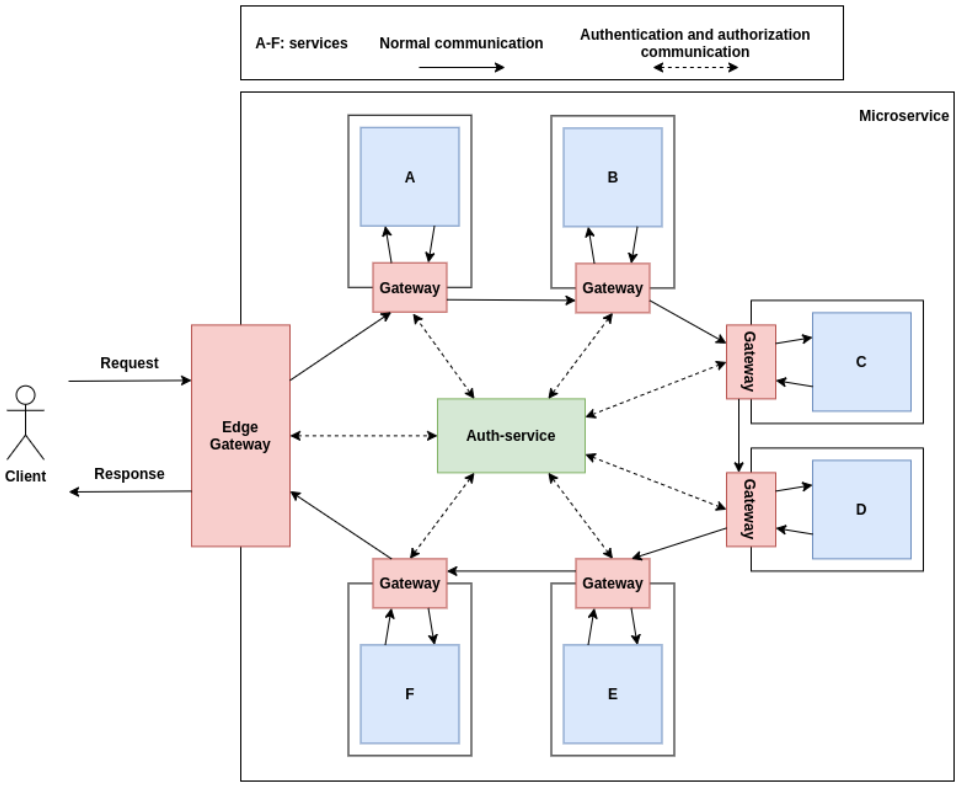

Figure 3.3: Visualization of the service level gateway pattern implementation in the microservice

Chapter 4

The testing framework

This chapter covers the two different types of tests performed. In Section 4.1 the research process which guided the testing procedure is covered. Section 4.2 describes how the security was tested. Section 4.3 details the load testing that generated the majority of the results.

4.1 Research process for load testing

The previous work that was cited in Section 2.6.1 guided the project in finding a suitable method to perform load tests for microservices in particular.

Since testing a security solution for vulnerabilities can be an entire project in itself this thesis opted to ensure that the security technologies used provided the advertised security. This ensured that the security solution had been implemented properly.

In order to ensure that both the hardware and software that the load testing software was running on was capable of handling the threads needed to simulate users its resource consumption was measured. The results were processed using the load testing software to get values such as the median and average. To analyse the data further Python scripts were used. The Python scripts used linear regression in order to show the trend of the data so that a prediction for how heavier loads than what was tested may result. The scripts were also used to create visual representations of the data as scatter plots but mainly as box plots in order to present a wider range of the data.

4.2 Security tests

The security tests looked at four scenarios which could occur through regular usage. These were:

- Using the microservice with valid credentials
- Omitting the token necessary for authorization
- Sending an invalid token
- Sending a valid token for a user that does not posses the correct role to access the requested resource

In order to test this the requests were sent directly to the API gateway located at the edge of the microservice. This was done using Postman* in order to change the `Authorization` header with ease and to display the results returned from the microservice. These tests were inspired by the security testing performed in the article by Nehme *et al.,* [28]. However, the tests mentioned in this section are not focusing on any specific attack such as token theft mentioned in the article. This is because the security provided in the project's microservice is only meant to provide general protection and more attention is on their effect on performance.

4.3 Load testing

This section describes the details of the testing framework developed for the load tests performed. Section 4.3.1 describes the structure of the framework. Section 4.3.2 describes the test cases used. Section 4.3.3 specifies the system used for testing and presents the CPU usage for different number of threads testing the microservice. Finally, Section 4.3.4 gives the parameters and other details of the testing.

4.3.1 Structure of the framework

Load testing with JMeter† was used to assess the impact which the three different security patterns had on the response times. For accurate load tests, the actual loads and service should ideally be as close to the tested service as possible, *i.e.,* the latency from the load generator should be as low as

* https://www.postman.com/ † https://jmeter.apache.org/

possible. Hence, a Spring Boot application dedicated for testing was deployed on the system under test, thus all communication was local to the cluster. This application used a maven package to programmatically run JMeter *JMX* files. The reason for having the application use Spring Boot was so that the files defining the test could easily be uploaded; as these files were often edited locally using the JMeter GUI. After uploading a *JMX* file the test results, which were in the format of a *JTL* file, was returned as a response from the Spring Boot application. These *JTL* files could then be viewed and analyzed using a locally running instance of JMeter. Postman was used to simplify this process of uploading files and retrieving the test results. Figure 4.1 displays a visual representation of where the service running JMeter is located and how its requests are sent into the microservice.

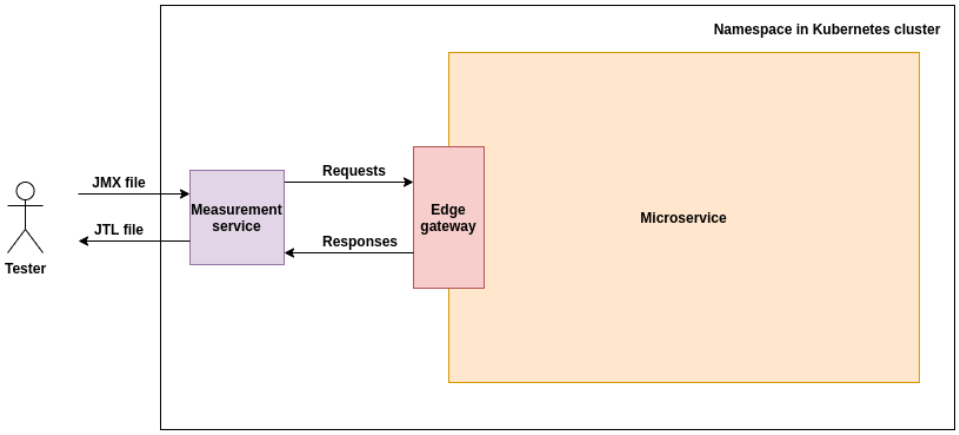

Figure 4.1: Graph displaying the tester and measurement service in relation to the microservice being tested

4.3.2 Test cases

The tests were divided into three different groups with one group of tests for each security pattern. In each group, the same test was run using an increasing number of JMeter threads to emulate multiple users sending requests to the system. The number of users ranged from 100 to 2000 with increments of 100 between each test run. To establish a baseline, an additional test run was also made with only one user.

As mentioned in Section 2.3, one of the main advantages of Kubernetes is the ease of scaling the components of a microservice. Therefore, it was of

interest to see if this scaling could be beneficial if performed on the different components of the microservice that implemented the security patterns. Since the different security patterns required a different setup of components (mainly due to the addition of different gateways) or how often certain services would receive a request, it was hypothesized that horizontal scaling of pods could make an impact on the response times. To asses the effects of scaling, the following components were scaled in different tests: (i) the auth-service, (ii) the gateways, and (iii) the normal services executing the business logic. This was done in order to be able to compare the effect of scaling each of the different types of components. The scaling was done by increasing the number of pods in a Kubernetes workload from one to three pods which was the most common number of replicas in the microservice. The load used was 2000 threads, *i.e.,* the heaviest load for the regular tests used when comparing security patterns. The reasoning behind this was to be able to more clearly see if any improvement was made in response times and that testing the entire range of thread counts would be to time consuming for the project.

4.3.3 System specifications and concurrency

Tables 4.1 and 4.2 display software and hardware specifications respectively. These are the specifications of the Kubernetes node that the Spring Boot application running JMeter load tests were deployed on. Note that one virtual CPU (vCPU) corresponds to an Intel® Xeon® GOLD 6148 model 64-bit processor running at 2.40 GHz. One such processor has four cores with one thread per core.

The service that was running an instance of the JMeter software used the specs shown in Tables 4.1 and 4.2. By using a specialized test with a plugin for measuring performance called *Servers Performance Monitoring* [*] in conjunction with a plugin measuring active threads over time [†] the resource usage was measured for different thread counts. Table 4.3 shows the CPU and RAM usage with 2000 active threads. Using this data it is possible to assert that JMeter and the platform it is deployed on is capable of running at least 2000 threads concurrently without depleting the CPU and RAM resources as both types of resources stay well below 80%.

[*] https://jmeter-plugins.org/wiki/PerfMon/
[†] https://jmeter-plugins.org/wiki/ActiveThreadsOverTime/

Table 4.1: Software specifications

Type	Specifications
Kernel version	4.15.0-128-generic
OS Image	Ubuntu 18.04.5 LTS
OS	Linux
Architecture	amd64
Container Runtime Version	containerd://1.2.13
Kubelet Version	v1.18.12
Kube-Proxy Version	v1.18.12

Table 4.2: Hardware specifications

Type	Specifications
CPU	Quad core vCPU
Memory (RAM)	16 GiB

Table 4.3: Active threads resource usage

Active threads	CPU usage %	RAM usage %
2000	55 - 67	28 - 37

4.3.4 Test parameters and details

The JMeter tests simulated the desired number of users as threads. The tests uses a ramp-up of 0 seconds which means that requests from threads are made as soon as JMeter is able, *i.e.,* without waiting for any results from previous threads. This is done in order to simulate a certain number of users waiting for their response from the microservice, *i.e.,* an open control loop is used. These threads concurrently wait for results; therefore, their level of concurrency is bounded by the capabilities of the underlying implementation. In JMeter's case, Java's thread model and scheduling policy together with the hardware determine the concurrency (for more details, see Section 4.3.3). Whenever the Kubernetes workloads were redeployed (*i.e.,* the auth-service, normal services, and the gateways) a dry run was performed to avoid the issue of a cold start were a newly started server could potentially have response times much higher than normal. Each request carried a header with a JWT which enables it to pass through the security checks of the gateways so that the entire MSA could be part of the tests.

Chapter 5

Results and Analysis

In order to ensure that the auth-service and gateways provided the intended protection three scenarios were tested which is covered in Section 5.1. The rest of the chapter displays the results from the load tests. It separates the results into two categories. The first category can be found in Section 5.2 and covers the response times as the thread count rises. The second category is found in Section 5.3 and includes the results when the heaviest load used in the non-scaled tests are horizontally scaled. Both these categories include the tests between the three different security patterns. Finally, Section 5.4 analyzes the validity of the work.

5.1 Testing the security

As the load tests focused on requests that are valid, it was important to ensure that the security solution also rejected requests considered invalid. To verify this the scenarios mentioned in Section 4.2 were tested. Figure 5.1 shows a successful request as the token is valid and the associated user has all roles required for authorization. As all services have received the request they have appended their secret data to the response (*i.e.,* a different Greek letter for each service). Figures 5.2 and 5.3 shows a similar response as when no or an invalid token is received it returns an error object with a reason. In this case, the reason is the roughly the same as no token is considered an invalid token. Figure 5.4 shows a slightly different error as the user associated with the sent token does not have the required authorization to reach beyond any gateways protecting the two last services.

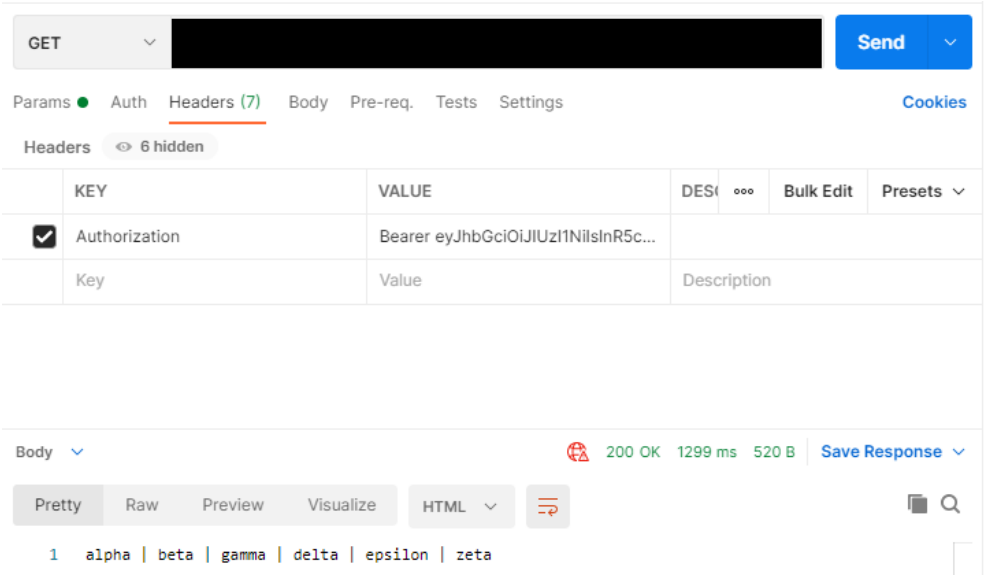

Figure 5.1: Postman response of a request bearing the correct JWT.

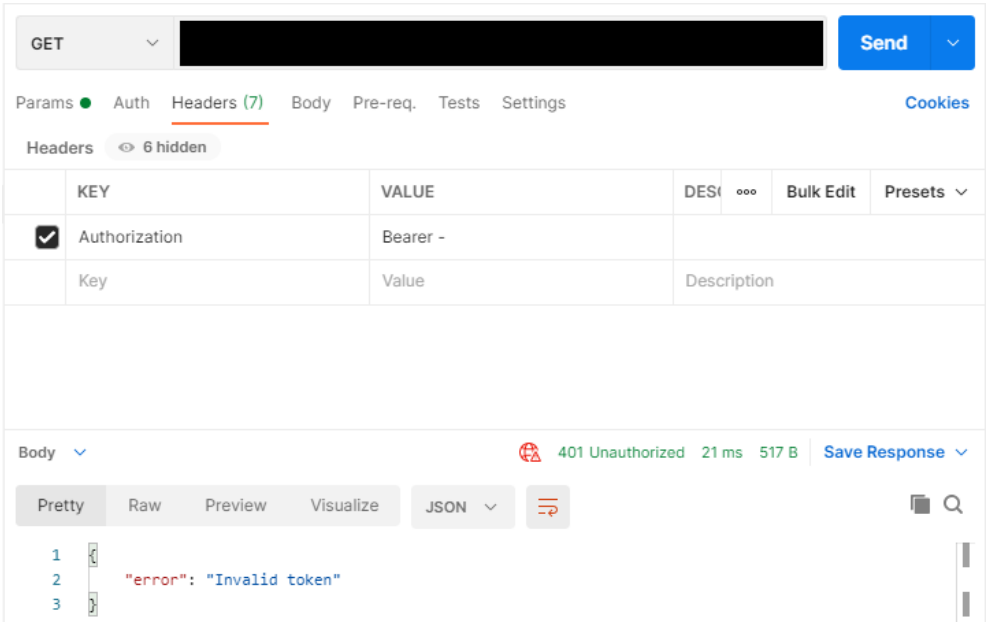

Figure 5.2: Postman response of a request bearing an invalid JWT

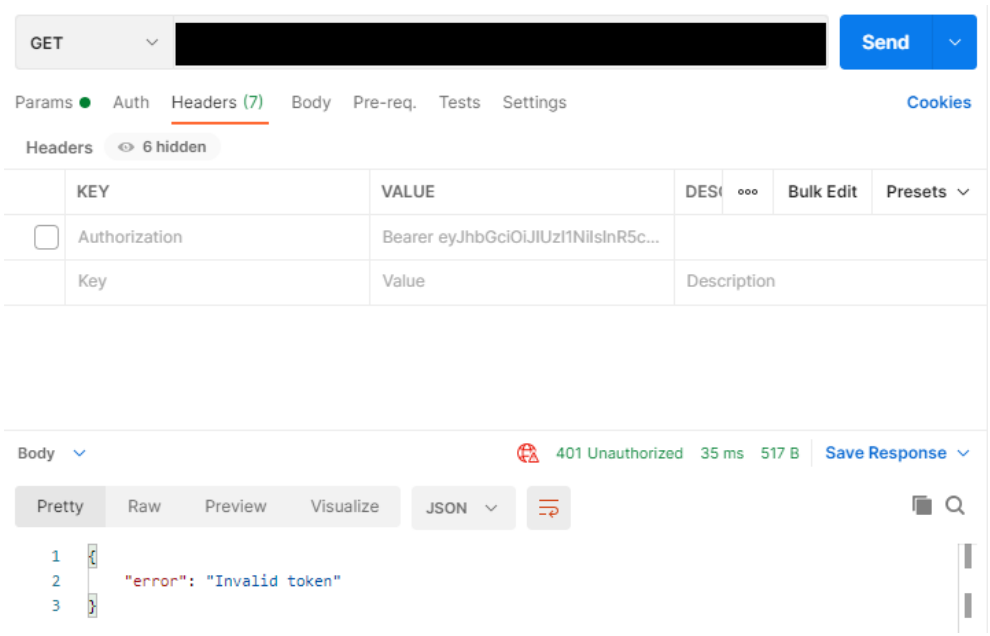

Figure 5.3: Postman response of a request omitting the JWT

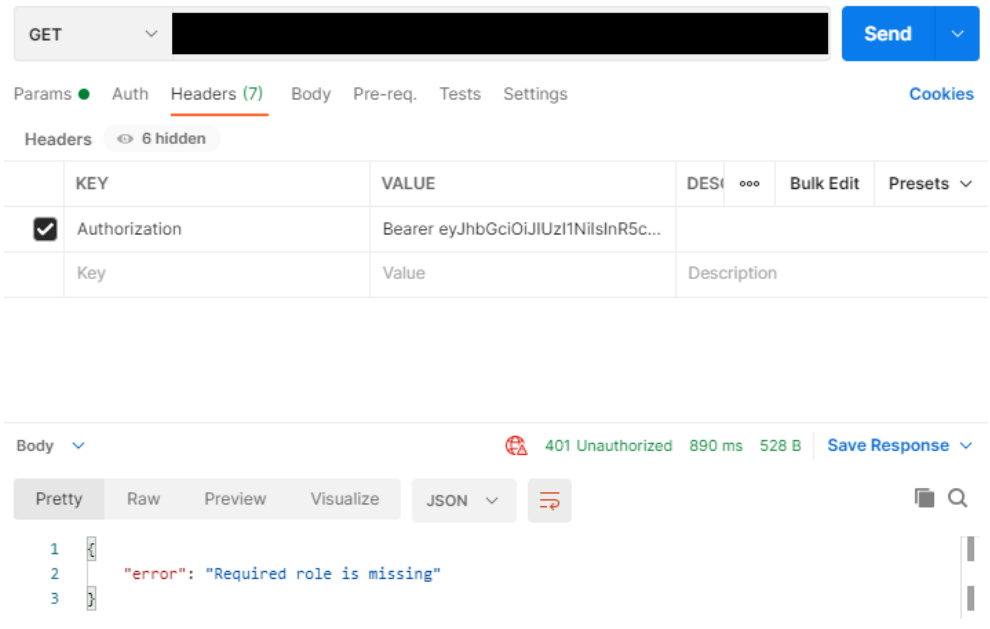

Figure 5.4: Postman response of a request bearing the correct JWT but lacking the necessary role for the last two services.

5.2 Load test results for increasing thread count

In this section, box plots and scatter charts show the results of the testing described in Chapter 4. The following subsections will examine each security pattern and present what results the load testing yielded when they were applied to the microservice. Figures 5.5 and 5.6 show the security pattern response times in direct comparison to each other in terms of the median and average response times (respectively). They reveal a steady increase in response times as the thread count increases across all security patterns. What is most noteworthy is that the *difference* in response time increases between the security patterns as the load increases. The service level gateway pattern is seemingly affected to a higher degree than the service group gateway pattern. This indicates a faster growth rate in response times for the service level gateway pattern as compared to the other two security patterns. Another interesting observation is that all patterns are almost equal in response times until the thread count reaches around 300-400 threads. A line has been fitted to the scatter plot's data points for each security pattern. The intercept is roughly equal for all lines with a difference of around 120 ms for the median and 26 ms for the average. The r^2 value for all fitted lines are 0.960 or above, indicating that it is likely that the increments in response times is tied to the thread count increase. With this it can be assumed that for higher thread counts the generated equations can make a good prediction as to what the resulting response time will be.

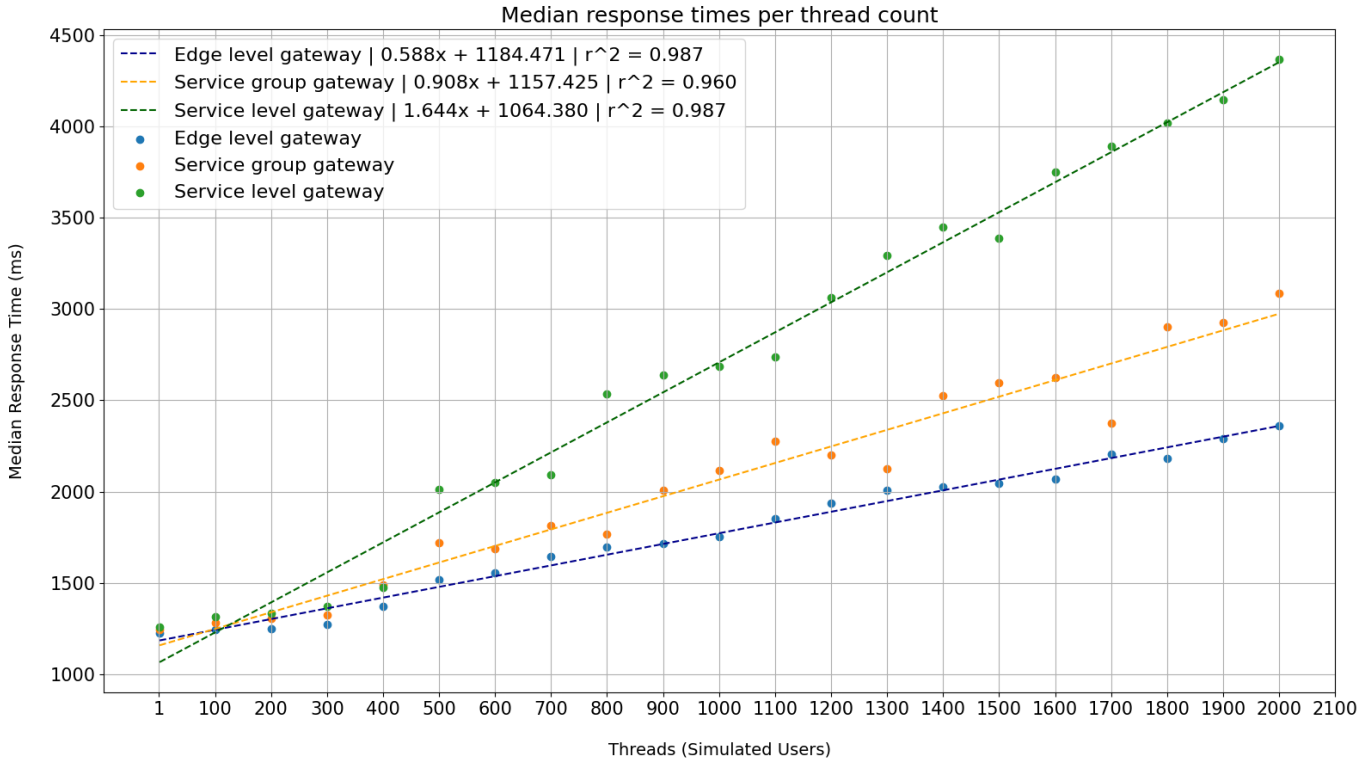

Figure 5.5: Median response time for all three patterns as the number of users (threads) increases. For each security pattern a line has been fitted through linear regression. The slope, intercept, and r^2 values are rounded to three decimals.

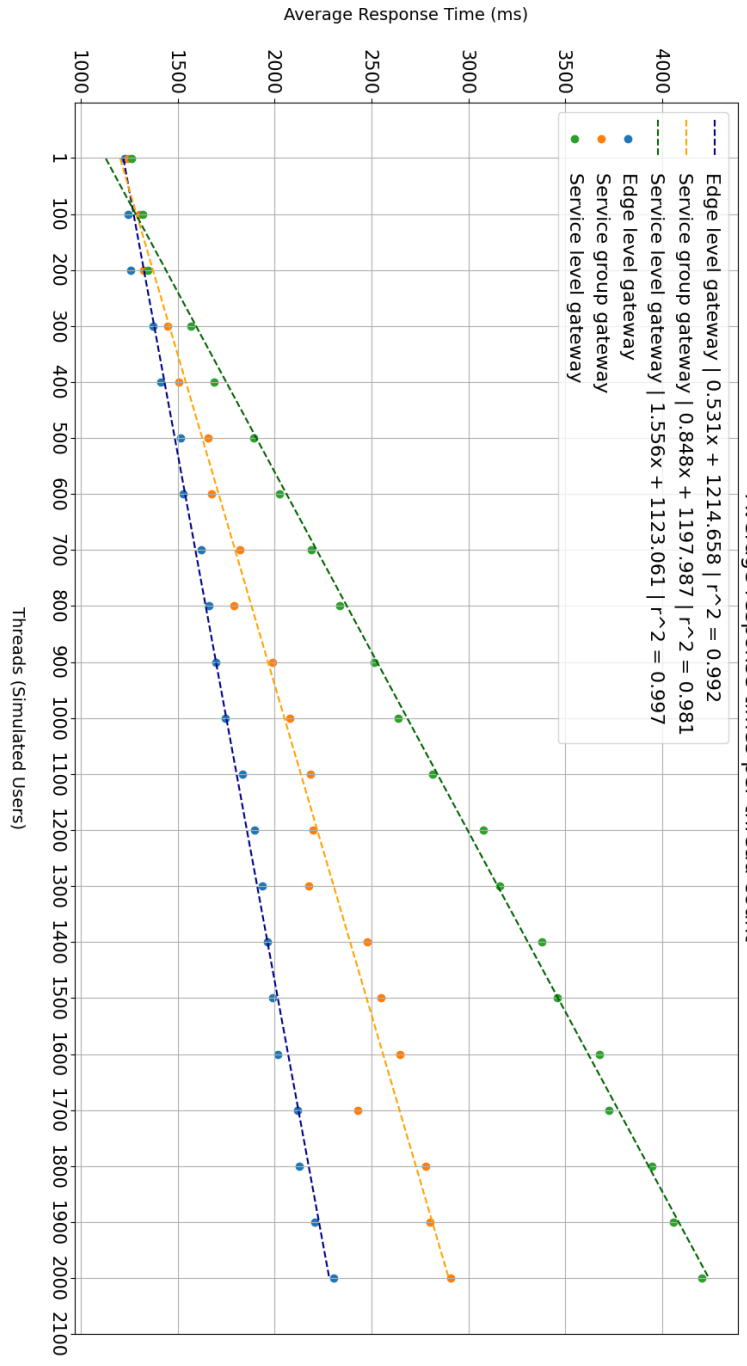

Figure 5.6: Average response time for all three patterns as the number of users (threads) increases. For each security pattern a line has been fitted through linear regression. The slope, intercept, and r^2 values are rounded to three decimals.

5.2.1 Edge level gateway response times

Figure 5.7 shows a moderate increase in response times using the edge level gateway pattern. As the value of the slope is 0.588 describes a rather careful increase of the median. When the thread count increases so does the size of the box and whiskers which indicates that the response times become more varied as their range is broadened. This is possibly due to the request backlog growing as the system can handle less and less users as JMeter has had time to create more threads which are sending their request to the microservice. Hence, the bottom whisker always stays at roughly the same position as the first requests can pass through the system without any other requests having to be served before them. As the top whisker is generally longer than the bottom one it suggests that outside the inter quartile range the higher response times are generally more varied the lower ones. However, as the thread count rises both whiskers become quite long in comparison to the box with a relatively centered median inside the box. This suggests a somewhat symmetric distribution apart from a longer tail towards the higher response times.

5.2.2 Service group response times

The service group gateway pattern is similar to the edge level pattern, as the response time gradually increases, as shown in Figure 5.8. What is noteworthy in this plot is how the positioning and length of the box, as well as the positioning of the median is less consistent. This is in comparison with the previous pattern (Section 5.2.1) and the following one (Section 5.2.3). Despite the amount of threads used is steadily increasing the median may dip below the value of the preceding test with 100 less threads and the box will in some places be larger as well as situated lower in comparison with its preceding box plot. This suggests a less predictable outcome in response times when the concurrent user count rises. This uncertainty in conjunction with how the open load test creates threads by only ensuring that threads make their request as soon as it is able, may have been the cause of these results. However, an overall upwards trend can still be spotted. This trend is as evident from the value in the slope, 0.908, a steeper increase in median response times compared to the edge level gateway patterns slope of 0.588.

5.2.3 Service level gateway response times

For the last pattern, Figure 5.9 shows a similar increase as in the previous security patterns. The plot shows a rather steady increase in box size

suggesting that as the thread count rises data becomes more the values are less centered around the median and have become dispersed. The slope for the median reveals not only the steepest slope of 1.644 but also a greater difference in slope from service level gateway to service group gateway pattern (a difference of 0.736) than from service group gateway to edge level gateway pattern (a difference of 0.320).

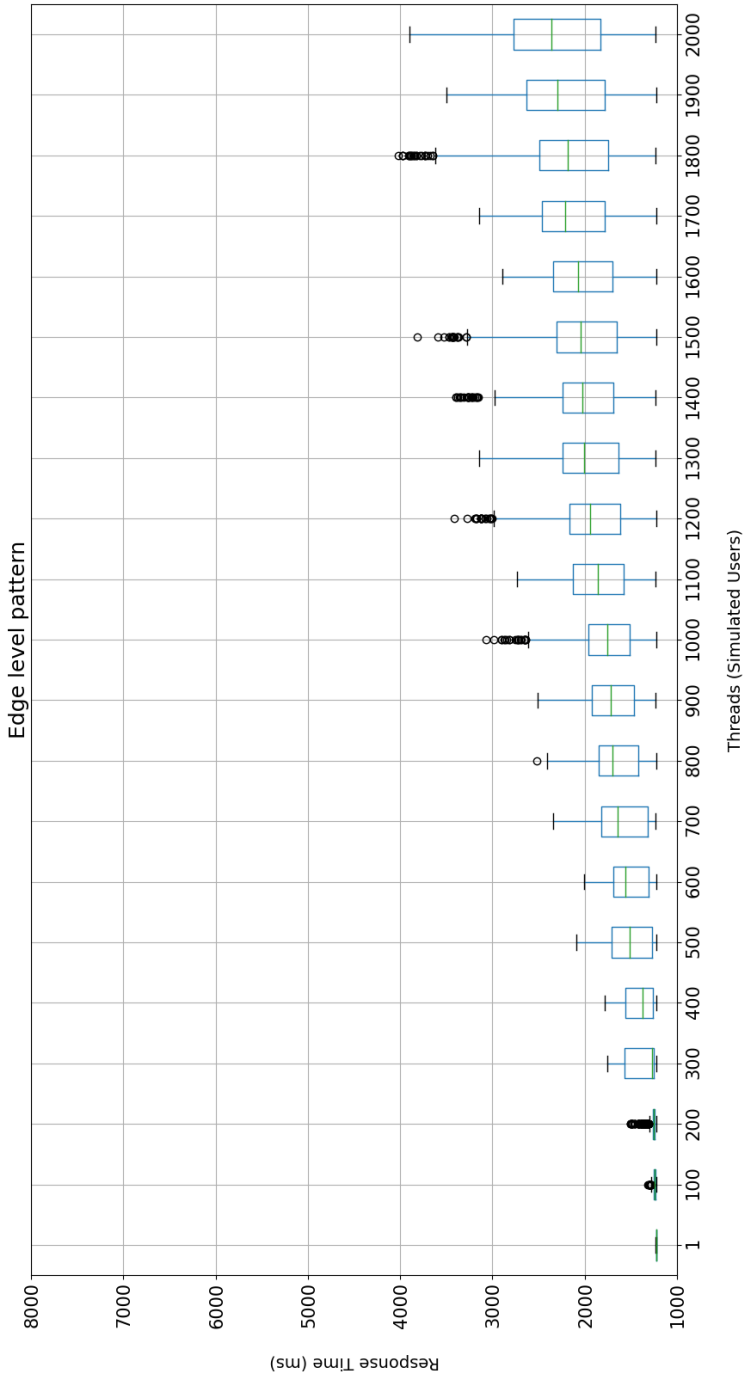

Figure 5.7: Box plot of edge level response time for different thread counts.

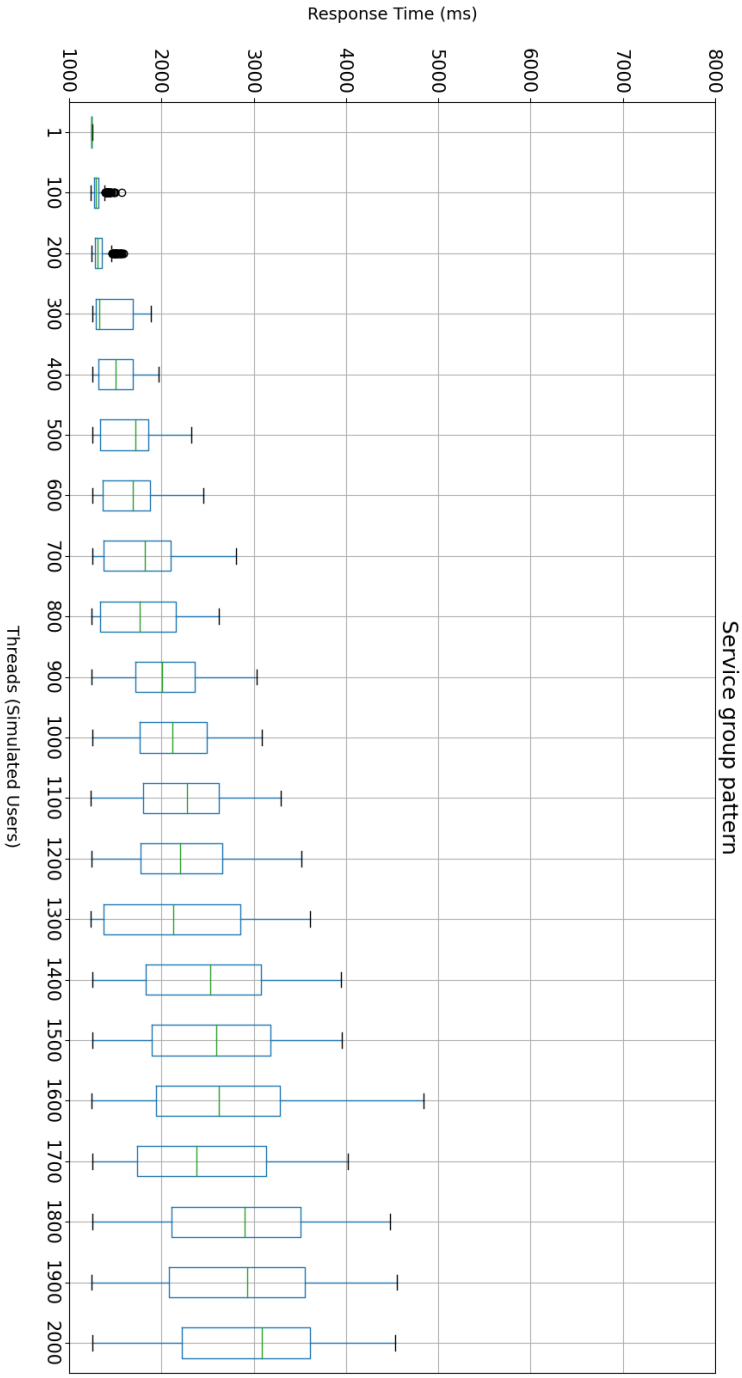

Figure 5.8: Box plot of service group response time for different thread counts.

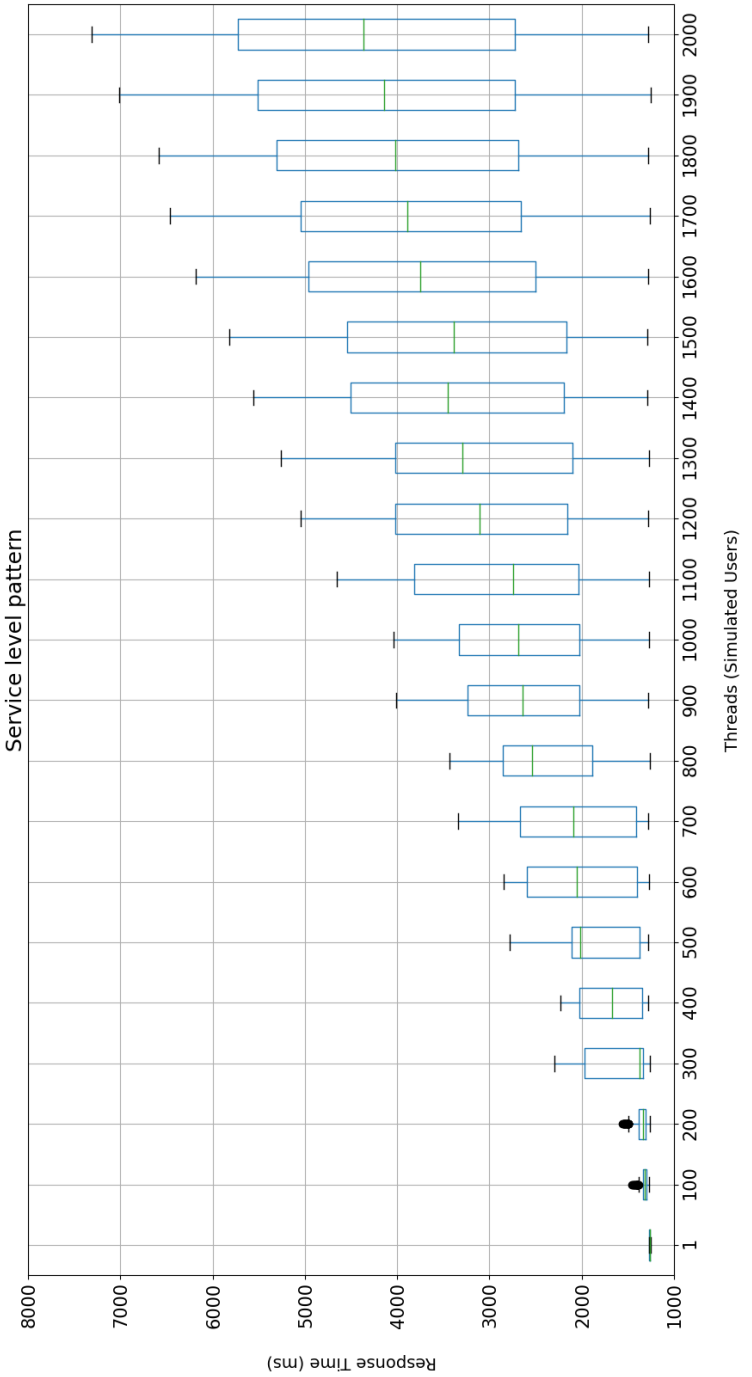

Figure 5.9: Box plot of service level response time for different thread counts.

5.2.4 Comparing the security patterns

Using Table 5.1 displays all median values for easier comparison. The increase in response times for the edge level gateway pattern went from 1225 ms for one thread to 2362 ms for 2000 threads. This is an increase of about 93%. For the service group gateway pattern the corresponding increase was 1244 ms to 3086 ms giving a 148% increase. Lastly, for the service level gateway pattern this increase was 1260 ms to 4367 ms meaning a 247% percentage increase. To compare the percentage increase of the 2000 threads load between the three security patterns there was a 31% increase from edge level to service group, 85% from edge level to service level, and finally 42% from service group to service level. Note that all percentages were rounded to the closest integer.

To visualize how the response time is distributed between each security pattern for every thread count, Figure 5.10 and continued in Figure 5.11 shows the all box plots for the aforementioned security patterns plotted together and grouped by thread count. This exposes a small difference in lowest response time which increases with how many gateways the security pattern makes use of. The service level gateway pattern's lowest response time is consistently higher than that of the two other ones and the service group gateway is higher than that of the edge level gateway. What is also worth noting is that the box size for the service level pattern becomes far greater in size than the other two patterns as the thread count is increased. This difference in size points to response times values may differ more from the median for security patterns of more security.

Table 5.1: Median response times for the security patterns.

Threads	**Edge level gateway response time (ms)**	**Service group gateway response time (ms)**	**Service level gateway response time (ms)**
1	1225	1244	1260
100	1243	1284	1315
200	1248	1306	1332
300	1272	1323	1371
400	1370	1487	1473
500	1516	1719	2013
600	1555	1687	2049
700	1644	1815	2094
800	1697	1765	2536
900	1715	2007	2639
1000	1752	2115	2687
1100	1854	2274	2740
1200	1937	2201	3065
1300	2006	2124	3292
1400	2026	2526	3447
1500	2047	2595	3388
1600	2067	2625	3750
1700	2207	2374	3892
1800	2181	2901	4017
1900	2292	2925	4145
2000	2362	3086	4367

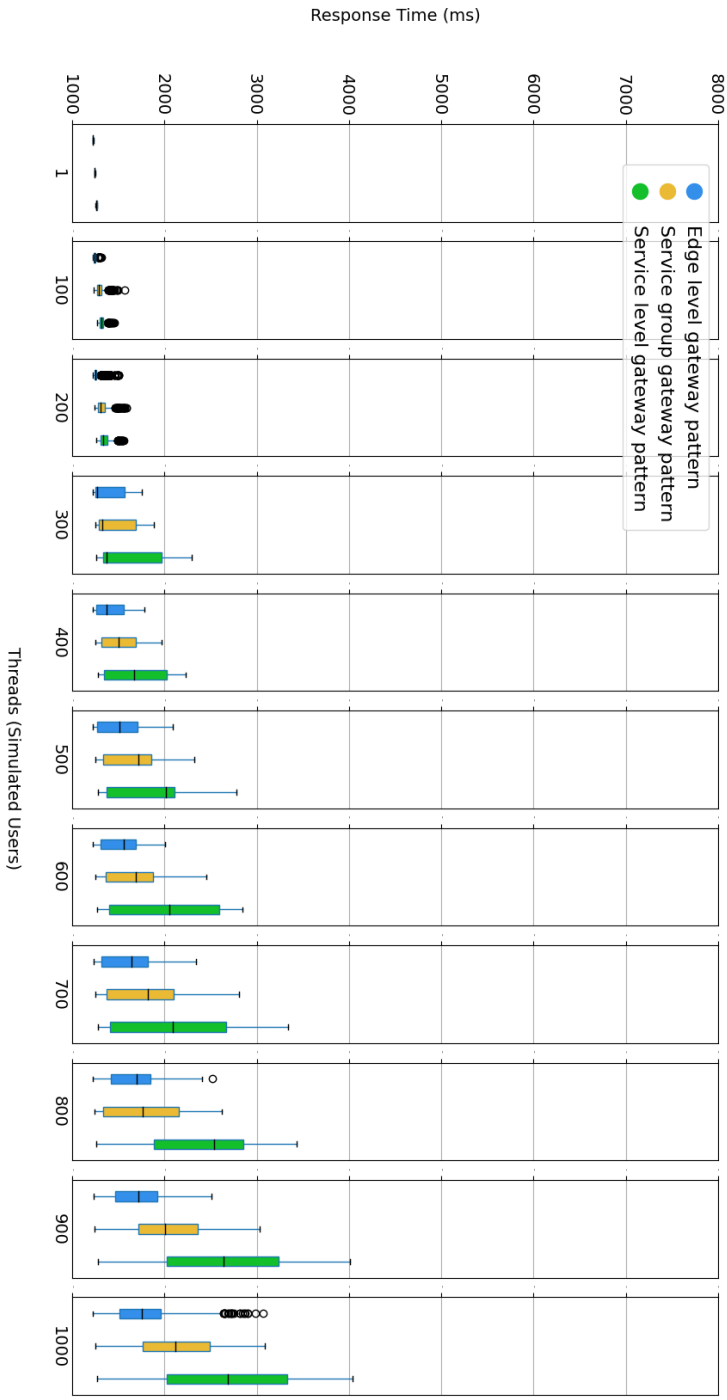

Figure 5.10: Box plot of all investigated security patterns for thread counts 1 to 1000.

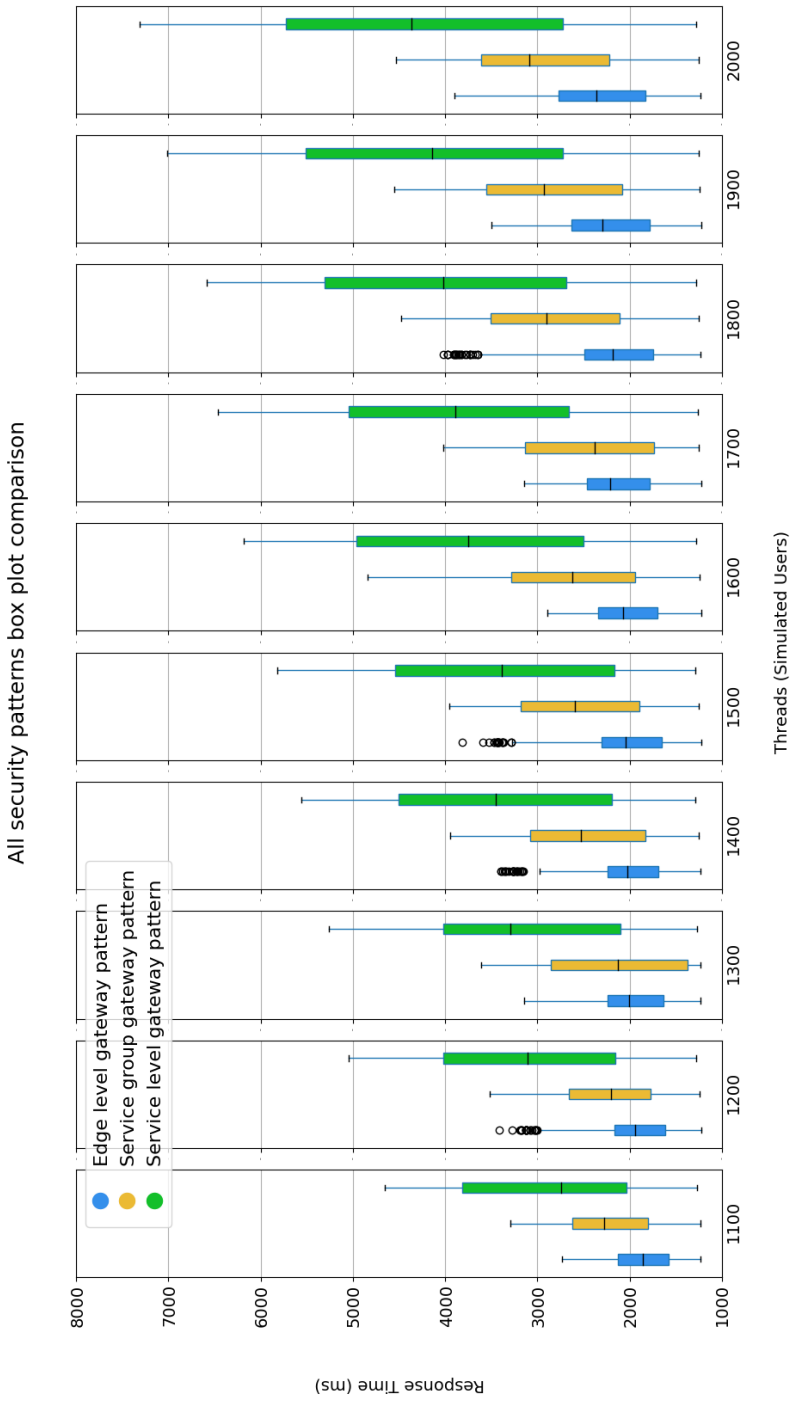

Figure 5.11: Box plot of all investigated security patterns for thread counts 1100 to 2000.

5.3 Scaling components of the microservice

This section presents the results when scaling different parts of the microservice. The results are split into the different security patterns and compared to the corresponding results when the same number of threads are run *without* scaling. The scaled components are either: the auth-service, the gateways, or the services.

5.3.1 Edge level pattern scaling comparison

Figure 5.12 shows that scaling had a small effect on reducing response times for the auth-service but not when scaling the gateway. Instead, when scaling the gateways the response times seemingly increased a small amount. Scaling the service greatly improved the performance compared to any other type of scaling.

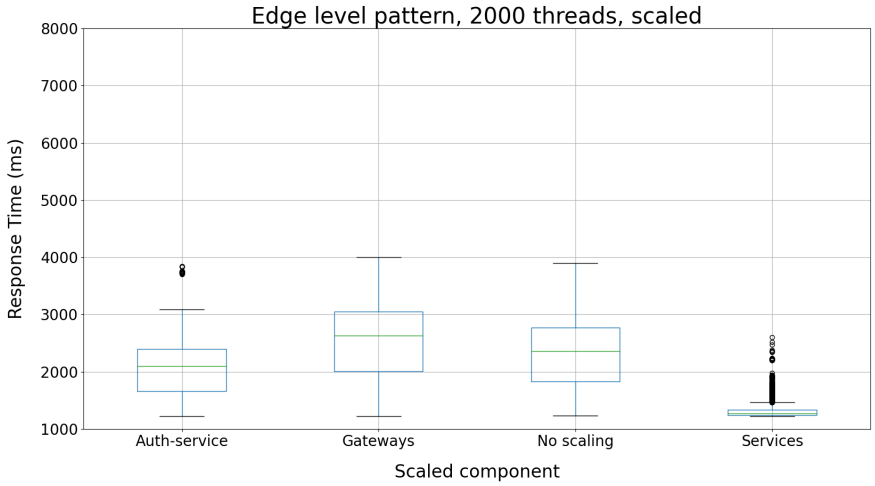

Figure 5.12: Box plots of scaling different components while implementing the edge level gateway pattern. Results for the same thread count with no scaling is included for comparison.

5.3.2 Service group pattern scaling comparison

For the service group pattern, each scaling test run improved the response time shown in Figure 5.8. Compared to the previous edge level pattern, the

effects are still not particularly strong but are slightly more visible. What is worth keeping in mind is that more components are affected by the scaling in this case — as this authentication and authorization pattern makes use of more gateways (and consequently grants more traffic to the auth-service). Still, scaling the services proved far more effective than the other forms of scaling.

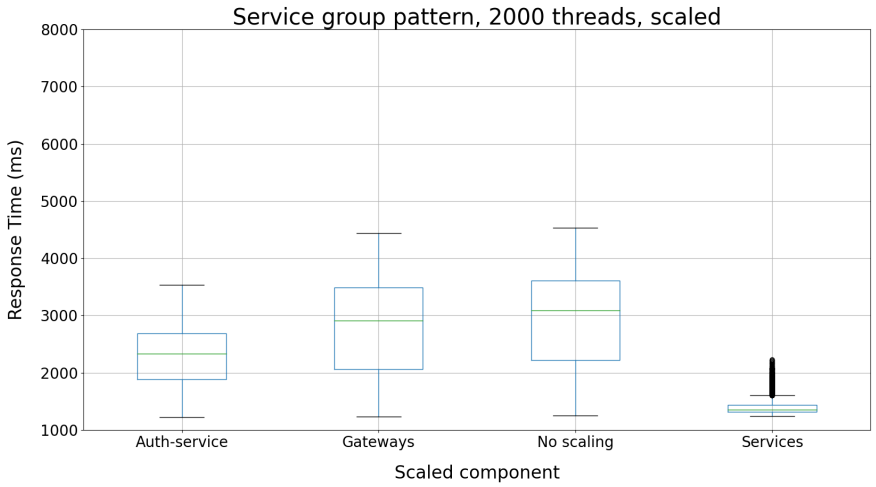

Figure 5.13: Box plots of scaling different components while implementing the service group gateway pattern. Results for the same thread count with no scaling is included for comparison.

5.3.3 Service level gateway pattern scaling comparison

Figure 5.14 shows the results of scaling different targets of the microservice when using the service level gateway pattern. These results show an increase in the relative benefit from scaling, further suggesting that scaling had at least some positive effect. Again there are more components that is affected by the scaling as compared to the two previous security patterns. This further suggests that increased complexity can lead to an increase in scaling benefits. As before scaling the services proved the most effective by far. However, scaling the auth-service also had a large effect - that was notably greater than the effect of this scaling for the two other patterns.

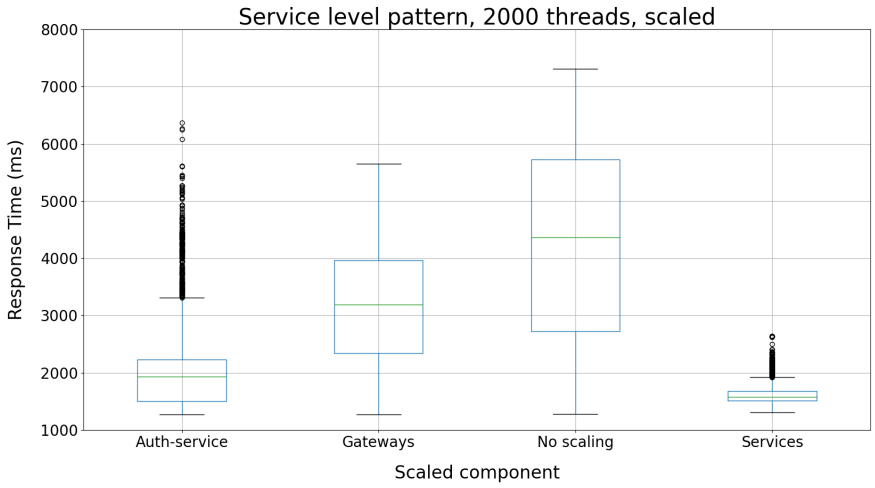

Figure 5.14: Box plots of scaling different components while implementing the service level gateway pattern. Results for the same thread count with no scaling is included for comparison.

5.3.4 Comparing the overall scaling results between security patterns

Comparing the overall benefit from scaling, the results showed that in general the service level gateway pattern benefited the most form being scaled. However, all patterns were greatly affected by the scaling of the services. Using the values in Table 5.2 the percentage increase between security patterns when scaling the services were as follows: 7% between edge level and service group, 25% between edge level and service level, and 16% between service group and service level. Contrast this with the corresponding percentages in Section 5.2.4 which were: 31%, 85%, and 42% respectively. This shows that the scaling drastically reduced the gap in performance between the security patterns. This further suggests that for microservice which either sees less concurrent users or has scaled the services in charge of performing time consuming work, a security pattern featuring more security-in-depth can be used while still retaining performance close to the edge level gateway pattern. The percentages have again been rounded to the nearest integer.

Figure 5.15 gives an overview on how the different security patterns responded to scaling compared against each other on the different components. It becomes more apparent that if the security pattern makes use of a component

more often and thereby increasing its work load, it responds better to scaling. This is best displayed with how well the service level gateway pattern responded to scaling the auth-service as compared to the other two security patterns. Something important to note is that the security patterns using internal gateways will have more scaled instances when scaling on the gateways. This is due to the fact that these gateways are deployed in the same way as individual services.

Table 5.2: Median response times in milliseconds for the security patterns when scaled on different components. Results for same thread count is included for comparison

Security pattern	**Auth-service**	**Gateways**	**No scaling**	**Services**
Edge level	2097	2634	2362	1266
Service group	2336	2913	3086	1357
Service level	1927	3194	4367	1579

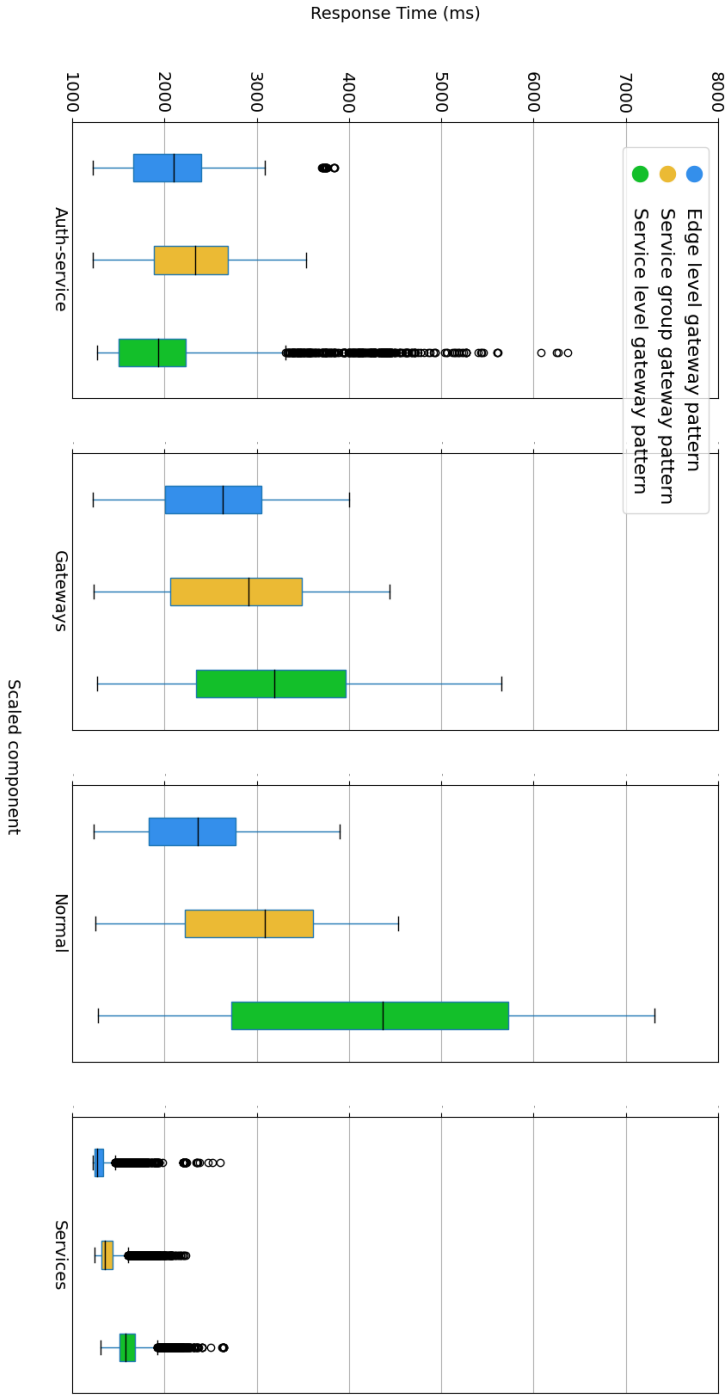

Figure 5.15: Box plot of all investigated security patterns showing the effects of scaling.

5.4 Validity Evaluation

All services and gateways used within the microservice are identical in implementation and functionality. This was done purposely, so that no response time would be different from any reason other than because the security patterns were different. In the evaluation of all patterns, the thread count was also varied in the same way. While services in a microservice are normally different based on the fact that they are meant to work towards a unique goal, it makes the analysis simpler to have homogeneous services. Even though it might seem like a better choice to try and create a replication of a specific scenario, there is simply too much variation in what a service could do; hence it would be difficult to have results consistent enough to generalize well. Therefore, this research opted for identical services and gateway implementations in a given pattern. Because of this, these results are more general and can be applied to more settings than a very specific case study; however, this come with a small trade-off in accuracy due to the increased generality.

JMeter can be used in a variety of ways, all of which have different benefits and drawbacks. Creating a test plan for use when testing a normal server or even microservice is easily be done via the JMeter GUI. While it may also seem simple to run the test directly from the GUI, this actually decreases the performance which is needed to run heavier load tests. What one should do instead, as was done in this thesis, is to run JMeter without the GUI. This can be done by running it from the command line or by directly running the Java code from another Java application as in the test setup described in Section 4.3.1. In addition, results returned from a JMeter test run can be captured directly and be presented in a comprehensible way by adding a listener. However, this also is another pitfall that drains too much of the system's performance for there to be accurate load tests. Instead, the results can be saved and added to a separate listener *after* a test has been run. The latest version of JMeter (as of writing this thesis is version 5.4.1) was also used so that all improvements are present. These are all part of JMeter's best practices*. The last measure taken for the load tests to be accurate was to place the application running JMeter as close to the tested backend as possible so that the response times are not skewed by anything causing overhead between JMeter and whatever application it wants to test.

Between five repetitions of the used load test the coefficient of variation had a value of less than five percent for the median results when running

* https://jmeter.apache.org/usermanual/best-practices.html

a standard load test against the microservice. For Ericsson this was an acceptable level of accuracy and since the literature study on load testing for MSA did not find any widely accepted standard this number of repetitions was settled upon. Since the process of analyzing and aggregating results using JMeter proved difficult to master and therefore tedious, it was difficult to estimate the time needed to complete and gather results. Even though the tests themselves were generally quick to complete (ranging from ~ 5 seconds to ~ 2 minutes) the amount of individual tests were quite high (70 in total). In addition, changes were often made to the tests since JMeter required some trial and error. This caused all tests to have to be redone as all tests needed to have been made using the same load test configuration. In order to avoid the tests consuming to much time, for the sake of finishing the thesis work on time, the amount of repetitions was not increased even though more repetitions could potentially have increased the accuracy (though the current level was deemed acceptable).

Chapter 6

Discussion

This chapter discuss the results and their analyses. This will be done in respect to, the results themselves (see Section 6.1), the goals mentioned (see Section 6.2), the research question (see Section 6.3), and the background (see Section 6.4).

6.1 Results

In order to decide upon what security pattern to recommend it needs to be defined what is seen as a positive result. The intuitive reasoning would be to look for a balance between a high level of security and low response times. However, despite a security pattern showing long response times relative to the other patterns it could still be seen as performing well relative to a reasonable threshold. Either a preference from the managers of the cluster which the microservice is deployed on or a widely recognized standard could be used. However, as there was no specific details regarding a goal for performance of the system this could not inform the conclusion of the best security pattern. As for general standards an article by Daniel An, former employee at Google, mentioned that less than three seconds is preferred [47]. While this could be used as a guideline the result comes from an investigation into public online services which does not quite fit the setting of the cluster used in the thesis project. Despite this, for a general conclusion it can be said that a good security pattern should not add more than three seconds of overhead to the response time while the system is experiencing common user counts.

The results show a rather clear trend where the median response times increase faster as the number of users increases and the greater the usage of gateways and the auth-service are. Interestingly, the response times are almost

equal for low numbers of users, suggesting that added overhead has negligible effect if the microservice is not expected to host a large number of concurrent users. Despite the difference in number of gateways used between the edge level gateway and service group gateway versus service group gateway and service level gateway patterns, the corresponding difference in response times is not equal. The service group gateway pattern has half the amount of internal gateways compared to the service level gateway pattern. As such it could be hypothesised that its response times would lie closer to the middle between the edge level and service level gateway patterns. However, it is instead closer to the edge level gateway pattern. This could possibly be due to the fact that in the edge level gateway pattern the flow of how the request is sent between services, gateways, and the auth-service can be seen as a straight line. First the edge gateway is reached which contacts the auth-service, subsequently each service is passed in a set order. In contrast, the service level pattern (and to a lesser extent the service group pattern) needs to reach the auth-service in between each request from one service that wishes to get a response from another service. This creates an increased number of IPC messages. Not only are the earlier requests blocking the later requests from entering services but later requests can block earlier requests form entering the auth-service. This creates a likely location for a bottleneck.

Something worth noting is that higher numbers of gateways in a security pattern leads to a greater range of response times when the thread count increases. This means that for the service level gateway pattern in particular the response times that users can expect may vary drastically. Instead of all users experiencing the same delays some will be affected worse than others which may be undesirable.

Looking at the results from scaling the different components of the microservice when the different patterns are applied leads to some interesting revelations. Scaling the gateways while using the edge level gateway pattern resulted in response times somewhat worse than not applying any scaling whatsoever. As the load test uses an open loop load generator this can create some variation in number of concurrent users. Since the benefits (and drawback) from scaling either the auth-service or gateways with the edge level pattern were quite small, there is a possibility that this security pattern does not benefit from scaling any of the components related to security. The increase in response time when performing the scaling on the gateway could then be due to the variation in load created by the open loop load generator. One can speculate that since the edge level pattern has fewer gateways and therefore less requests leading to the auth service, scaling these components would not

have a great effect. In contrast, scaling of the security patterns using internal gateways proved more beneficial. Since the number of gateways increases (and by extension the usage of the auth-service) the total work done by these components increases. Meaning that scaling can have a greater impact on these components. Apart from scaling the services, scaling the auth-service seems to have been the most efficient choice. As mentioned the auth-service can become a bottleneck for request flow; hence, scaling the auth-service assisted in reducing response times. Scaling the services proved to consistently have the greatest benefit across the three security patterns. As the services simulated work taking 200 ms it makes sense that scaling the service itself would be a prime candidate for horizontal scaling. Something that is worth mentioning is that in order to be able to scale a component this will require additional resources. This means a monetary cost for the proprietor of the system hosting the microservice. Since this thesis does not take into account the cost of scaling the pods that make up its microservice there *may* be a limit to the feasibility of the scaling performed.

6.2 Meeting the goals

The goals were to create a security solution that was capable of authentication and authorization which in its base form can be seen as the edge level gateway pattern. The rest of the security patterns implemented could also be seen as possible security solutions for usage by Ericsson. The measurements could then guide them in the selection of the most appropriate security pattern. This also ties into the completion of the second goal which was to cover areas of MSA security not previously researched or that needed further work. As Section 2.4 noted, there was a clear lack of published scientific articles focusing on the security patterns. This was met by contributing to this area and expanded upon existing research by comparing three patterns for authentication and authorization.

6.3 Answering the research question

In Section 1.2.3, the research question asks what an appropriate security pattern for a microservice would be and how it would compare to other patterns. The literature study found that, while common, an effective and well studied strategy was to issue tokens for authentication and use either roles or attributes at a later stage if authorization was needed. This was shown

to be possible wherever API gateways were found, as they made a logical security checkpoint. To the best of the authors knowledge, research using these gateways to achieve "security-in-depth" by placing multiple internal gateway instances throughout a microservice was a more obscure idea and therefore researched on a more individual level. This made it a prime candidate for comparison with other setups for gateway authentication and authorization which then became the different security patterns studied in this thesis.

When attempting to secure a system, it is often a better idea to use technology proven to be capable of providing the protection sought. This philosophy was followed when choosing technologies for the security patterns, resulting in the usage of JWT, role-based authorization, and API gateways. While this gave a sense of the level of security, it was of interest to infer some measure of the difference in security between the set of patterns that were implemented. To achieve this the attack vector where services could be controlled by a malicious actor was used. Each secured connection from service to service was then seen as an increase in security. The pattern for service level gateways was identified as the most secure option.

In terms of performance versus security trade-off, the service group gateway pattern can be seen as the best choice. It gives increased security but has a less rapid increase in overhead as the number of users increases. It can also be observed in Figure 5.10 and Figure 5.11 that up until 1700 threads the added overhead in terms of delays does not exceed three seconds as mentioned in Section 6.1. For heavier loads the worst response time is still not much greater than the three second threshold. However, it should be stated that this is only a general suggestion and that it is likely better to use all the results to come to a conclusion best suited for securing an individual microservice. For the specific case regarding the Kubernetes cluster used in the project there are less users accessing it at any time than the range of 300 to 400 users. Higher numbers of users beyond this range is where the patterns starts to diverge in terms of performance and additional authentication and authorization leads to an increase in median response times. Therefore, there is no reason not to recommend the security pattern delivering the most amount of security as the results show that the effect on the performance will be almost equal to that of the pattern with the least amount of security. As such the recommended security pattern for the microservice used in the project is the service level gateway pattern.

6.4 Relating to the background

The load tests performed in this thesis drew inspiration from the performance tests in the article by Akhan Akbulut and Harry G. Perros [41]. While it is not said explicitly, this is an example of an open workload model or at the very least it is described as one. The hope is that this will give the two works some amount of comparability and set a trend for future work so that it too will have results comparable to previous work.

In the article by Safaryan, *et al.,* [25] the suggested implementation was one of the main sources of inspiration to what would become the most basic layer of security: the edge level gateway pattern. The article claims that their system would be able to handle heavier loads when the amount of users on the system increases. However, the article did not provide any specific metrics or other results to confirm this claim. This thesis is able to both test this claim and also puts it into a deeper context. As it was shown that the security pattern corresponding with the article's proposed implementation can be scaled if done correctly. The scaling in this case should then be performed on the services handling the business logic for there to be any noticeable effect.

The work by Nehme, *et al.,* [28] provided the inspiration for the service level gateway pattern. However, there were some key differences in implementation details. Nehme, *et al.,* substituted an edge level gateway for each service. In contrast, the security patterns in this thesis opted for a singular point of entry which meant keeping the edge level gateway. Another difference was in the number of services and the authorization method used. Nehme, *et al.,* opted for XACML while this thesis used a simpler role based approach. Keeping these difference in mind, the overhead of the method proposed by Nehme, *et al.,* was calculated to be less than 32% for 250 service calls. In contrast, the service level gateway pattern used in this thesis had an increase of 7% and 8% (rounded to an integer) for 200 and 300 threads respectively. This is possibly due to the additional security features provided by the security solution implemented in the microservice used by Nehme, *et al.,* as it supported usage of multiple tokens and XACML but could also be due to a difference in technologies used to implement the microservice.

Chapter 7

Conclusions and Future work

This chapter provides a conclusion to this thesis report from different aspects. Section 7.1 provides general conclusions directly tied to the work, results, and analysis. Section 7.2 brings up what limitations affected the process and results. Some possible and suggested future work is covered in Section 7.3. Lastly, Section 7.4 addresses some ethical, economical, and environmental issues through some reflections by the author.

7.1 Conclusions

This section offers some conclusions starting in Section 7.1.1. Section 7.1.2 describes the evaluation. Section 7.1.3 addresses some issues that arose during this thesis. Section7.1.4 offers some insights gain during this thesis. Finally, Section 7.1.5 offers some suggestions for others working with microservices.

7.1.1 Positive effects and drawbacks

In terms of what security solution can be applied using any of the three suggested security patterns, we assume that any security will at the very least require authentication and authorization to reach the microservices from outside the cluster. Additionally, all users must exist within the Ericsson LDAP server and have the correct roles needed to access certain parts of the microservice (if the security pattern used supports roles). Any internal protection may have the drawback of increased response times when the system is heavily loaded but the usage of additional gateways behind the edge of the microservice will provide a more granular approach to role authorization within the microservice. These measures required there to

be more communication with the auth-service, thus creating a potential bottleneck in the communication flow. As the results in Section 5.2 show, increasing layers of security increases the response times. When looking at the results a compromise between security and performance was always kept in mind. While no definitive conclusion can be made as to which security pattern is better than the other, it is possible to reason about in what context any of the given patterns should be used. For example, a system that has a heavy load but does not handle sensitive information will probably benefit from the edge level gateway pattern. This is because with increasing numbers of concurrent users there is not as dramatic an increase in response times as when the security pattern had more internal gateways communicating with the auth-service. In contrast, when security is of the utmost concern then performance may be secondary, especially if there have been previous instances or concerns about attackers taking control of internal services. In reality things may not be perfectly black and white. There may be a need to balance or compromise between performance and security. The obvious suggestion would then be the service group gateway pattern. Results showed that in terms of response times this security pattern was closer to the edge level gateway pattern (which was the fastest of the three) despite having half the number of internal gateways as compared to the service level gateway pattern. It is important to remember that the service group gateway is the most dynamic of the three security pattern and the presented results might differ from the results of another implementation. This variability comes from how many services are placed behind the same gateway. Services that are grouped together have the benefit of communicating without additional auth-service requests. This has the benefit of giving the software architect more options so that some services can be given a higher level of protection while others can be optimized for communication performance. However, this comes with a cost of increased time spent on defining coordination between services and what category of security they should be grouped by.

A positive effect of using gateways, particularly in regards to the ones internal to the microservice, is that developers of regular services are allowed to focus on the business logic. They are not forced to maintain their service's security needs and instead a dedicated team can oversee the gateways as they are identical in implementation (mentioned in Section 3.4). This means that the benefit of the MSA where teams can work independently on services is not affected.

As there is now data on different security patterns, there will hopefully be future work that can use this research to legitimize their claims. This not

only affects research that continues this particular thesis' work but also other projects in the same or adjacent areas.

An important drawback to consider is that the additional gateways means additional deployments in the cluster. As an example the service level gateway pattern requires one Node.js application which serves as the gateway for a certain service. This essentially doubles the number of deployments in a microservice. This can lead to increased complexity and resources used. As complexity is something the MSA strives to avoid, the need for additional gateways may be an unattractive choice for the architecture designers when deciding on how to implement their desired security.

7.1.2 Description of evaluation

The evaluation of a pattern's security was based upon how many points of communication were covered by an authorization using the auth-service. This did not rely on measured results but rather simply the findings from the literature study where more common security technologies could be seen as more trustworthy. This is in contrast to the performance evaluation, where the metrics from the literature study provided the basis for experimentally evaluating the impact on performance of each pattern. The fitted lines helped to evaluate how closely the performances of the security patterns were to each other in order to detect if there was an equal difference from pattern to pattern. The box plots evaluated the performance on the range of values which determined if users would experience similar response times.

7.1.3 Addressing issues

Early on in the load testing it was noticed that no matter how much load was used and whatever security pattern was active, the response times were always roughly the same. In addition, they were always exceedingly low, somewhere around 20 ms. The realization which solved this problem was that the services themselves which were meant to be as general as possible did not contribute much to the overall request service time since they did not do any actual task. These extremely low response times meant that the open loop load generator of JMeter could not create threads faster than previous threads received responses. This meant that the microservices never received a high enough load to create a backlog of requests. To combat this a simulated work time was added to each service in the form of a delay achieved by using Java's sleep function. When this was introduced, then response times rose as

the thread count did. JMeter proved to have another issue which was due to the difference between open versus closed loop load generation - as closed loop load generation waits for responses before sending the next request or requests. Since this is not a realistic scenario where one user would wait for another, JMeter was configured to send the requests as soon as possible (*i.e.,* as an open loop load generator).

Initially the ingress objects served as the gateways by redirecting requests meant for a service to the auth-service for authorization. However, certain tests would sometimes cause the microservice to become unresponsive (*i.e.,* it crashed). As the ingress objects were provided out-of-the-box by Kubernetes and configured using *YAML* files debugging was difficult. To solve this, the custom gateways detailed in Section 3.4 were deployed as nodes in the cluster.

7.1.4 Insights

The literature study that was conducted as a part of this thesis gave a lot of insight into the state of MSA security. While not surprising, it was interesting to see that the greatest volume of research articles dedicated to security were centered around the more practical concepts, such as authentication and authorization which this thesis also centers around. This is understandable, if research is associated with some corporation, then the goals and research question would be of a more pragmatic nature.

Another insight was that much research relied on security solutions similar to the edge level gateway pattern being sufficient to secure the microservice. This was surprising since some of the articles covering attack vectors and more complex security architectures often claimed that a singular point of security was not sufficient.

7.1.5 Suggestions for others in the field of microservice security

Some suggestions for others who wish to design architecture and develop components for security in microservices is to always keep in mind two things. Firstly, maximum security is **not** always feasible. The reality is that higher security can have negative effects on performance. Therefore, it is important to know what work the system is tasked to perform, so that the overhead does not become a burden. The overhead can be decreased by identifying attack vectors and designing specific protection rather than something that attempts to cover all possible attacks. This makes it possible to consider security-performance

trade-offs. Secondly it is of value to know what the security requirements are and if these requirements differs throughout the microservice. Ideally one should group together components in need of the same level of security measures as this was shown to lead to smaller increments in response times when the number of users increases.

7.1.6 What could have been handled differently

If there was a need to start over from the beginning with all lessons learned the main difference would possibly be in the choice of load testing tool. While JMeter proved feature rich and had an acceptable Java API which enabled it to be run from a Java program, it was quite difficult to master. The tool itself had a wide range of options most with ambiguities around the option's functionality. This meant that much trial and error as well as sanity checks were needed to fully grasp how to use JMeter for the desired test case. Even though JMeter may be more familiar to me now, a more modern load tester or even tests written directly in an appropriate programming language (although this may lead to some possible sources of error if the code is not thoroughly tested) would be preferable.

7.2 Limitations

Since the load test used an open loop load generator it has the drawback of it being more ambiguous how many users are concurrently inside the microservice. This ambiguity was seen as an acceptable trade-off since a closed loop variant only executes the next thread or set of threads when the previous ones receive their response, removing some of the load from the microservice and stopping the build up of a request backlog.

The security technologies used were limited to what was identified in the literature study as some of the more common examples. The motivation was to instead focus on the architectural features of the three security patterns studied rather than to attempt to use more esoteric means of performing authentication and authorization.

One limitation regarding validity is the Kubernetes cluster. Since it is not a cluster created solely for the testing of the patterns but rather for the use of these patterns in an existing Ericsson cluster this can have an effect on how general the results are. The microservice within the greater cluster shared this space with other services. However, to provide some isolation it was setup

in its own namespace. It was decided that setting up a separate Kubernetes cluster with a cloud provider was outside the scope of this thesis.

7.3 Future work

Suggestions for future work have been divided into consideration of other security technologies (Section 7.3.1), more complex internal gateways (Section 7.3.2), corporate vulnerability assessment (Section 7.3.3), and the use of audit trails (Section 7.3.4.)

7.3.1 Other security technologies

While role based access control, an OAuth 2.0 inspired authorization flow, and JWT tokens served as the main security technologies explored in this thesis, it could be interesting for the sake of innovation to assess whether other less used technologies could be applied while retaining one of the three presented patterns. Such potential future studies could either use JWT alternatives or use something entirely different, such as cryptographic keys or XACML. If these technologies proves more resource consuming, then it will then be of interest to investigate their scalability since it was demonstrated that a higher work load responds better to scaling. Again this could either be performed comparatively between patterns or have a focus on a selected pattern, preferably one which supports internal gateways — as this is less common. Another possibility is to compare the performance of state-of-the-art security technologies to these less broadly employed ones.

7.3.2 Complex usage of internal gateways

The security patterns presented in this thesis provide an additional layer of security. Both literally being an additional gateway that a request may need to pass but also in the sense that it can require a specific role to allow access. The next step could be to expand the gateways such that *e.g.,* a service group protected by one internal gateway could have another set of services inside it protected by a separate gateway. Another possibility is to have less homogeneous groups and a set of services residing directly behind the edge level gateway. As in this thesis the main metrics that could be extracted is the effects on performance.

7.3.3 Corporate vulnerability assessment

It may be of interest to perform a vulnerability assessment on an instance where any of the three security patterns are implemented in a microservice. As new vulnerabilities are continually exposed, it is important to be aware if the security solution is affected. Michał Walkowski, *et al.,* [48] proposed a vulnerability management centre. As this proposed system runs in a "dockerized" environment it could easily be deployed to the same cluster as the microservice running any of the three proposed security patterns. An investigation can then be performed into how this vulnerability management centre can be integrated with one or more of the security patterns and how well vulnerabilities are found.

7.3.4 Audit trails

No literature covering audit trails for microservices was found. What was found was older research by Theresa Lunt [49] detailing different aspects of audit trails. One possibility is to allow an algorithm to analyse the activity of users so that the process can be automated. The auth-service used with the service level gateway pattern would be a good position to generate audit trails as all communication is forced to pass through the auth-service. Each request could then be coupled with a user by associating it with the JWT located in the header. Lunt also claims that this could serve as the basis for an Intrusion Detection System (IDS), if automated.

7.3.5 What has been left undone?

There could have been more statistical tests performed between the different patterns to assert if there was a statistically significant difference in their performance. The main reason for not performing this was due to time constraints. It may have been more feasible of considered early but the literature study did not find such tests in previous works as seen in Section 2.6.2, Section 2.6.3, and to some extent in Section 2.6.1. Therefore, it was not considered in time. These articles mainly propose one suggestion to securing a microservice and do not attempt any comparisons which makes it understandable that they do not perform any test to see if their security solution is significantly better than any other. However, in the case of load testing different design patterns for the MSA it can be argued that a statistical test would have been of interest. Another possible reason for there not being any statistical tests in the related work and to some extent in this project it can be

argued that these types of tests are better suited when there is a possibility for more randomness in results and a need to determine if the differences are due to chance. While this might be suited when researching effects of medicine or results linked to human behaviour where there are more uncertainty testing machines lead to more predictable results. However, as seen in the results for scaling the edge level gateway pattern it did result in gateway scaling proving less beneficial which was not a predicted effect. In this case it could have been of interest to investigate if there was a statistical significance of the effects on response times from scaling components used in this security pattern.

7.3.6 Next obvious things to be done

What should be done next is to replace the delay with a function that performs arithmetic operations or some another predetermined set of operations which will make use of the services' CPUs. This can either be done for a certain amount of time or until the calculations are complete. This way the effect of resource starvation will be taken into account by the response times.

As explained in Section 7.3.5 there could be more focus on statistical tests. This would be done to ensure that there is a significant difference in security patterns' inherit benefits and drawbacks on performance.

7.4 Reflections

The login system can be argued to be the most important part of any application reachable by multiple people on a network. Therefore, it can also be argued that there is a considerable responsibility to ensure that it is implemented properly. As such, the implementation of the security technologies were not created as custom implementations to avoid introducing any critical bugs. An example is the JWT creation and verification, as there were already packages ready for usage which had been broadly implemented and tested.

When stronger security is implemented, this often leads to less accessibility. A simple and common example is the enforcement of stronger passwords which means that users are likely prompted to create passwords which are harder to remember and might have to change them every so often. This can make usage of the protected application less accessible, especially to those who have some form of mental disability. It is worth mentioning that the security patterns which were mentioned all require a JWT issued after a login, *i.e.,* it is subject to this problem. However, the restrictions on passwords can

be decided separately, as most of the relevant parts of the implementation are hidden from users.

Many examples of security implemented by articles covered in the literature study mention using a single gateway located at the edge of the microservice and propose that this is adequate security or even a good security solution. In contrast to this, the literature study also uncovered that there are security threats not covered by this level of security. This raises the question of whether or not it is dangerous to attempt to expose these security solutions as inadequate. The reasoning for publishing these results and findings is that the microservices found in the literature study are implementations created to demonstrate the efficacy of the security suggested in the articles. Irregardless, it should also be assumed that any malicious actors already posses this information; therefore, it is better to use these revelations as warnings so that proprietors of microservice who lack adequate security have the chance to improve upon it, possibly using the suggestions of this thesis as inspiration.

An ethical dilemma comes from the portrayal of the patterns with lesser security. If the difference in response times are far too large to consider some of the more complex patterns, then it might discourage system architects from attempting to implement them. Therefore, it was important that the background covered the benefits of "security-in-depth" and a selection of attack vectors with their related consequences. This could then serve as a warning to urge readers to push for better security despite some performance penalties.

References

[1] C. Richardson, *Microservice Patterns: With Examples in Java*. Manning Publications Co., 2019. ISBN 978-1617294549 [Pages 2, 10, 12, 13, and 18.]

[2] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah, "Securing Microservices," *IT Professional*, vol. 21, no. 1, pp. 42–49, 2019. doi: 10.1109/MITP.2018.2876987 [Pages 2, 10, 18, 19, and 23.]

[3] R. Chandramouli, "Security strategies for microservices-based application systems," National Institute of Standards and Technology, Gaithersburg, MD, USA, Special Publication (NIST SP) 800-204, Aug. 2019. [Online]. Available: https://www.nist.gov/publications/security-strategies-microservices-based-application-systems [Page 3.]

[4] A. Pereira-Vale, E. B. Fernandez, R. Monge, H. Astudillo, and G. Márquez, "Security in microservice-based systems: A Multivocal literature review," *Computers & Security*, vol. 103, p. 102200, 2021. doi: 10.1016/j.cose.2021.102200 [Pages 3, 4, 17, 19, and 23.]

[5] A. J. Jafari and A. Rasoolzadegan, "Security patterns: A systematic mapping study," *Journal of Computer Languages*, vol. 56, p. 100938, 2020. doi: https://doi.org/10.1016/j.cola.2019.100938 [Page 3.]

[6] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3, pp. 301–310, 2017. doi: https://doi-org.focus.lib.kth.se/10.1007/s00450-016-0337-0 [Page 10.]

[7] B. Benatallah and H. R. Motahari Nezhad, *Service oriented architecture: Overview and directions*. Berlin, Heidelberg: Springer, 2008, pp. 116–130. ISBN 978-3-540-89762-0 [Page 10.]

[8] T. Yarygina and A. H. Bagge, "Overcoming Security Challenges in Microservice Architectures," in *2018 IEEE Symposium*

on Service-Oriented System Engineering (SOSE), 2018. doi: 10.1109/SOSE.2018.00011 pp. 11–20. [Pages 10 and 22.]

[9] N. Chondamrongkul, J. Sun, and I. Warren, "Automated Security Analysis for Microservice Architecture," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020. doi: 10.1109/ICSA-C50368.2020.00024 pp. 79–82. [Page 10.]

[10] W. Jin, R. Xu, T. You, Y. G. Hong, and D. Kim, "Secure Edge Computing Management Based on Independent Microservices Providers for Gateway-Centric IoT Networks," *IEEE Access*, vol. 8, pp. 187 975–187 990, 2020. doi: 10.1109/ACCESS.2020.3030297 [Pages 12, 18, and 26.]

[11] H. David Booth, Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web Services Architecture," W3C Working Group Note 11 February 2004 https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest, W3C, Tech. Rep., 2004, accessed: 2021-02-15. [Page 13.]

[12] T. Yarygina, "RESTful is not secure," in *International Conference on Applications and Techniques in Information Security*. Springer, 2017, pp. 141–153. [Pages 13 and 14.]

[13] J. Reschke, "The 'Basic' HTTP Authentication Scheme," *Internet Request for Comments*, vol. RFC 7617 (Proposed Standard), Sep. 2015. doi: 10.17487/RFC7617. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7617.txt [Page 14.]

[14] Auth0, "Introduction to JSON Web Tokens," https://jwt.io/introduction, accessed: 2021-02-22. [Page 14.]

[15] R. Xu, W. Jin, and D. Kim, "Microservice Security Agent Based On API Gateway in Edge Computing," *Sensors*, vol. 19, no. 22, 2019. doi: 10.3390/s19224905 [Pages 14, 18, and 27.]

[16] "OAuth Core 1.0 Revision A," https://oauth.net/core/1.0a/, accessed: 2021-03-08. [Page 15.]

[17] D. Hardt, "The OAuth 2.0 Authorization Framework," *Internet Request for Comments*, vol. RFC 6749 (Proposed Standard), Oct. 2012. doi: 10.17487/RFC6749. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6749.txt [Page 15.]

[18] "OAuth 2.0," https://oauth.net/2/, accessed: 2021-02-15. [Page 15.]

[19] D. Yu, Y. Jin, Y. Zhang, and X. Zheng, "A survey on security issues in services communication of microservices-enabled fog applications," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 22, p. e4436, 2019. doi: https://doi.org/10.1002/cpe.4436 [Page 15.]

[20] M. Jones, B. Campbell, and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants," *Internet Request for Comments*, vol. RFC 7523 (Proposed Standard), May 2015. doi: 10.17487/RFC7523. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7523.txt [Page 15.]

[21] Y. ShuLin and H. JiePing, "Research on Unified Authentication and Authorization in Microservice Architecture," in *2020 IEEE 20th International Conference on Communication Technology (ICCT)*. IEEE, 2020. doi: 10.1109/ICCT50939.2020.9295931 pp. 1169–1173. [Page 15.]

[22] "Welcome to OpenID Connect," https://openid.net/connect/, accessed: 2021-03-04. [Page 15.]

[23] A. Pereira-Vale, G. Márquez, H. Astudillo, and E. B. Fernandez, "Security mechanisms used in microservices-based systems: A systematic mapping," in *2019 XLV Latin American Computing Conference (CLEI)*. IEEE, 06 2019. doi: 10.1109/CLEI47609.2019.235060 pp. 01–10. [Page 17.]

[24] A. Bánáti, E. Kail, K. Karóczkai, and M. Kozlovszky, "Authentication and authorization orchestrator for microservice-based software architectures," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2018. doi: 10.23919/MIPRO.2018.8400214 pp. 1180–1184. [Page 17.]

[25] O. Safaryan, E. Pinevich, E. Roshchina, L. Cherckesova, and N. Kolennikova, "Information system development for restricting access to software tool built on microservice architecture," in *E3S web of conferences*, vol. 224. EDP Sciences, 2020. ISSN 2267-1242 p. 01041. [Pages 18, 26, and 69.]

[26] X. He and X. Yang, "Authentication and Authorization of End User in Microservice Architecture," *Journal of Physics: Conference Series*, vol. 910, p. 012060, oct 2017. doi: 10.1088/1742-6596/910/1/012060 [Pages 18 and 26.]

[27] Q. Nguyen and O. F. Baker, "Applying Spring Security Framework and OAuth2 To Protect Microservice Architecture API," *Journal of Software*, vol. 14, no. 6, pp. 257–264, 2019. doi: 10.17706/jsw.14.6.257-264 [Pages 18 and 27.]

[28] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah, "Fine-grained access control for microservices," in *International Symposium on Foundations and Practice of Security*. Springer, 2018. doi: https://doi.org/10.1007/978-3-030-18419-3 pp. 285–300. [Pages 18, 26, 38, and 69.]

[29] A. V. Uzunov, E. B. Fernandez, and K. Falkner, "Securing distributed systems using patterns: A survey," *Computers & Security*, vol. 31, no. 5, pp. 681–703, 2012. doi: https://doi.org/10.1016/j.cose.2012.04.005 [Page 18.]

[30] K. Jander, L. Braubach, and A. Pokahr, "Defense-in-depth and Role Authentication for Microservice Systems," *Procedia Computer Science*, vol. 130, pp. 456–463, 2018. doi: 10.1016/j.procs.2018.04.047 The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops. [Pages 22, 24, and 26.]

[31] C. Otterstad and T. Yarygina, "Low-Level Exploitation Mitigation by Diverse Microservices," in *6th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, ser. Service-Oriented and Cloud Computing, F. D. Paoli, S. Schulte, and E. B. Johnsen, Eds., vol. LNCS-10465. Oslo, Norway: Springer International Publishing, Sep. 2017. doi: 10.1007/978-3-319-67262-5_4 pp. 49–56, part 2: Microservices and Containers. [Pages 22 and 23.]

[32] X. Li, Y. Chen, and Z. Lin, "Towards Automated Inter-Service Authorization for Microservice Applications," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3342280.3342288. ISBN 9781450368865 p. 3–5. [Page 22.]

[33] D. Richter, T. Neumann, and A. Polze, "Security Considerations for Microservice Architectures," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, INSTICC. SciTePress, 2018. doi: 10.5220/0006791006080615. ISBN 978-989-758-295-0 pp. 608–615. [Page 22.]

[34] M. Ahmadvand, A. Pretschner, K. Ball, and D. Eyring, "Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework," in *Federation of International Conferences on Software Technologies: Applications and Foundations*. Springer, 2018, pp. 573–588. [Page 23.]

[35] T. Yarygina and C. Otterstad, "A Game of Microservices: Automated Intrusion Response," in *Distributed Applications and Interoperable Systems*, S. Bonomi and E. Rivière, Eds. Cham: Springer International Publishing, 2018. doi: https://doi.org/10.1007/978-3-319-93767-0_12. ISBN 978-3-319-93767-0 pp. 169–177. [Page 23.]

[36] Y. Sun, S. Nanda, and T. Jaeger, "Security-as-a-service for microservices-based cloud applications," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015. doi: 10.1109/CloudCom.2015.93 pp. 50–57. [Page 24.]

[37] "Get Started with JSON Web Tokens," https://auth0.com/learn/json-web-tokens/, accessed: 2021-03-03. [Page 24.]

[38] A. Pashalidis and C. J. Mitchell, "A taxonomy of single sign-on systems," in *Information Security and Privacy*, R. Safavi-Naini and J. Seberry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN 978-3-540-45067-2 pp. 249–264. [Page 24.]

[39] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra, "Formal analysis of saml 2.0 web browser single sign-on: Breaking the saml-based single sign-on for google apps," in *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering*, ser. FMSE '08. New York, NY, USA: Association for Computing Machinery, 2008. doi: 10.1145/1456396.1456397. ISBN 9781605582887 p. 1–10. [Page 24.]

[40] C. Fan., A. Jindal., and M. Gerndt., "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application," in

Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER, INSTICC. SciTePress, 2020. doi: 10.5220/0009792702040215. ISBN 978-989-758-424-4. ISSN 2184-5042 pp. 204–215. [Page 25.]

[41] A. Akbulut and H. G. Perros, "Performance Analysis of Microservice Design Patterns," *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019. doi: 10.1109/MIC.2019.2951094 [Pages 25 and 69.]

[42] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016. doi: 10.1109/ETFA.2016.7733707 pp. 1–6. [Page 27.]

[43] D. Yi, T. Donghui, W. Xuegang, and Z. Shuliang, "Application of Authorization in Smart Grid based on the PasS Microservice architecture," *IOP Conference Series: Earth and Environmental Science*, vol. 512, p. 012118, jun 2020. doi: 10.1088/1755-1315/512/1/012118 [Page 27.]

[44] "Welcome to Kong Docs," https://docs.konghq.com/, accessed: 2021-05-31. [Page 27.]

[45] D. Kallergis, Z. Garofalaki, G. Katsikogiannis, and C. Douligeris, "CAPODAZ: A containerised authorisation and policy-driven architecture using microservices," *Ad hoc networks*, vol. 104, p. 102153, 2020. doi: https://doi.org/10.1016/j.adhoc.2020.102153 [Page 27.]

[46] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "CBOR Web Token (CWT)," *Internet Request for Comments*, vol. RFC 8392 (Proposed Standard), May 2018. doi: 10.17487/RFC8392. [Online]. Available: http://www.rfc-editor.org/rfc/rfc8392.txt [Page 27.]

[47] Daniel An, "Find out how you stack up to new industry benchmarks for mobile page speed," https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/, accessed: 2021-06-21. [Page 65.]

[48] M. Walkowski, M. Krakowiak, J. Oko, and S. Sujecki, "Efficient Algorithm for Providing Live Vulnerability Assessment in Corporate Network Environment," *Applied Sciences*, vol. 10, no. 21, 2020. doi: 10.3390/app10217926 [Page 77.]

[49] T. Lunt, “Automated audit trail analysis for intrusion detection,” *Computer Audit Update*, vol. 1992, no. 4, pp. 2–8, 1992. doi: https://doi.org/10.1016/0960-2593(92)90034-K [Page 77.]

TRITA-EECS-EX-2021:390