MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES IN PURE AND APPLIED SCIENCES

DESIGN OF A QUEUE-BASED MICROSERVICES ARCHITECTURE AND PERFORMANCE COMPARISON WITH MONOLITH ARCHITECTURE

KENAN CEBECİ

MASTER THESIS

Department of Computer Engineering

ADVISOR

Assist. Prof. Ömer KORÇAK

ISTANBUL, 2019

MARMARA UNIVERSITY
INSTITUTE FOR GRADUATE STUDIES
IN PURE AND APPLIED SCIENCES

DESIGN OF A QUEUE-BASED MICROSERVICES ARCHITECTURE AND PERFORMANCE COMPARISON WITH MONOLITH ARCHITECTURE

KENAN CEBECİ

(524111011)

MASTER THESIS

Department of Computer Engineering

ADVISOR

Assist. Prof. Ömer KORÇAK

ISTANBUL, 2019

MARMARA UNIVERSITY

INSTITUTE FOR GRADUATE STUDIES IN PURE AND APPLIED SCIENCES

Kenan CEBECİ, a Master of Science student of Marmara University Institute for Graduate Studies in Pure and Applied Sciences, defended her thesis entitled "**Design of A Queue-Based Microservices Architecture and Performance Comparison with Monolithic Architecture**", on June 21, 2019 and has been found to be satisfactory by the jury members.

Jury Members

Assist. Prof. Dr. Ömer KORÇAK (Advisor)
Marmara University .....

Assoc. Prof. Dr. Murat Can GANİZ (Jury Member)
Marmara University .....

Assist. Prof. Dr. Ali NİZAM (Jury Member)
Fatih Sultan Mehmet Vakif University .....

APPROVAL

Marmara University Institute for Graduate Studies in Pure and Applied Sciences Executive Committee approves that Kenan CEBECİ be granted the degree of Master of Science in Department of Computer Engineering, Computer Engineering Program on 07.08.19 (Resolution no: 2019/16-02).

Director of the Institute **Prof. Dr. Bülent EKİCİ**

ACKNOWLEDGMENT

I would like to express my gratitude to my thesis supervisor, Assist. Prof. Ömer Korçak, for his guidance, support and encouragement throughout my graduate study and completion of this thesis. I would also thank to Assoc. Prof. Murat Can Ganiz and Assist. Prof. Ali NİZAM for participating my thesis committee and their useful comments.

I would like to thank Assist. Prof. Gökay Burak Akkuş, Assist. Prof. Yaşar Safkan and İdil Gülnihal Sağlam for their support and friendship. I would like to Ramazan Çamcı , Onur Doğan and Ozan Tek for their helps during the measurements.

Finally, I would like to thank my family for their patience, encouragement and support during my whole life.

July, 2019 **Kenan CEBECİ**

TABLE OF CONTENTS

TABLE OF CONTENTS iii
ÖZET v
ABSTRACT vi
SYMBOLS vii
ABBREVIATIONS viii
LIST OF FIGURES x
LIST OF TABLES xii
1. INTRODUCTION 1
1.1. Developer Operations (DevOps) 2
1.2. Cloud Computing 3
1.3. Software Architectures 5
1.3.1. Monolith 6
1.3.2. Microservices 7
1.4. Related works and motivation 8
2. METHODOLOGICAL CONSIDERATION 11
2.1. Architectural Evaluation 11
2.1.1. Monolithic architecture 11
2.1.2. Service-based architectures 13
2.1.3. Service-oriented architecture versus microservices 15
2.1.4. Monolith versus microservices 16
3. PROPOSED DESIGN OF MICROSERVICES ARCHITECTURE 21
3.1. Communication 23
3.1.1. API gateway 24
3.1.2. Messaging Data Format Selection 28
3.1.3. Inter-microservices communication 28
3.2. Service Registry and Discovery 30
3.3. Modularity 35
3.4. Security 36
3.4.1. Authentication and Authorization 37
3.5. Database Selection 39
4. EXPERIMENTAL RESULTS 41
4.1. Test Environment 41

4.2. Performance of The Implemented Prototype Application 42
4.3. Database Performance of Monolith and Proposed MSA 45
5. CONCLUSION AND FUTURE WORK 48
REFERENCES 49
RESUME 1

ÖZET

KUYRUK TABANLI BİR MİKROSERVİS MİMARİSİ TASARIMI VE MONOLİTİK MİMARI İLE KARŞILAŞTIRILMASI

Kurumsal bir yazılım sisteminin oluşturması veya dönüşümü, iş ihtiyaçlarının tam olarak tanımlanmasını gerektiren meşekkatli bir işlemdir. İş gereksinimlerinin karşılanabilmesi için iyi düşünülmüş, uygun yazılım mimarisi kararlaştırılmalı ve tasarlanmalıdır. Genel olarak sorunlara çözüm bulmak için takip edilebilecek iki yöntem vardır. Birincisi geleneksel monolitik mimaride olduğu gibi problemi, doğru çözümü bulmak için bir bütün olarak ele almak. İkincisi ise problemi daha kolay anlaşılabilen ve çözülebilen küçük parçalara ayırmaktır. Eğer yazılım dünyasında ikinci yöntem takip edilecek olursa, mikroservis mimarisi gündeme gelmektedir. Kurumsal ölçekli yazılım sistemi tasarlanmak istendiğinde, bildiğimiz kadarıyla yazılım mimarilerini değerlendiren, iletişim protokolü, veri modeli ve veritabanının seçimini üzerine yol gösterici deneysel bir araştırma bulunmamaktadır. Bu tezde, kolay ölçeklenebilir, bakım yapılabilir, erişilebilirliği yüksek, güvenilir ve gözlemlenebilir mikroservis tabanlı bir yazılım sistemi tasarlanmıştır. Ayrıca amacına uygun yazılım mimarisi ve modellerini seçmeye yardımcı olabilecek şekilde farklı mimarilerin, iletişim protokollerinin ve veri modellerinin karşılaştırıldığı deneysel çalışmalar sunulmuştur. Tüm makale sadece sunucu servis tasarımı ile ilgili olup istemci tipi ve teknolojileri bu çalışmanın kapsamı dışındadır.

July, 2019 **Kenan CEBECİ**

ABSTRACT

DESIGN OF A QUEUE-BASED MICROSERVICES ARCHITECTURE AND PERFORMANCE COMPARISON WITH MONOLITH ARCHITECTURE

Building or transformation of an enterprise software system is an onerous process which requires precise definition of business demands. Then to enable the satisfaction of business requirements, the well-thought-of and convenient software architecture must be determined and designed. According to common sense, there are two methods to be followed in order to find the right solution for a problem. One is to handle the problem as a whole; like the traditional monolith architecture. The second method is to divide the problem into easily understandable and soluble fine-grains. If the second path is chosen in software world, the microservices architecture can be shown. When the entire enterprise level system design is considered, to the best of our knowledge, there is no any leading empirical research on the evaluation of software architectures, selection of communication protocol, data formats, and database. In this thesis, an easily scalable, maintainable, highly-available, reliable and observable software system is designed by comparing variant architectures, communication methods, and data models that would help to choose the most appropriate architecture or model for the right purpose. All the thesis is about designing a backend API system. The client types or technologies are out of scope.

July, 2019 **Kenan CEBECİ**

SYMBOLS

ABBREVIATIONS

CSE : Continuous Software Engineering

CPU : Central Processing Unit

DevOps : Developer Operations

SDLC : Software Development Lifecycle

QoS : Quality of Services

OS : Operating System

IaaS : Infrastructure-as-a-service

HaaS : Hardware-as-a-service

PaaS : Platform-as-a-service

SaaS : Software-as-a-service

API : Application Programming Interfaces

SOA : Service Oriented Architecture

DDD : Domain Driven Design

SRP : Single Responsibility Principle

MSA : Microservices Architecture

REST :Representative State Services

RDBMS : Relational Database Management System

EA : Enterprise Architecture

SoC : Separation of Concerns

XML : Extensible Markup Language

JSON : Java Object Notation

ACID : Atomicity, consistency, isolation and durability

PoC : Proof-of-Concept

IoT : Internet of Things

AI : Artificial Intelligence

IPC : Inter-Process Communication

SOAP : Simple Object Access Protocol

WSDL : Web Services Description Language

AMQP : Advanced Message Queuing Protocol

ESB : Enterprise Service Bus

JWT : Json Web Token

LIST OF FIGURES

Figure 1.1 A standard monolith architecture design 6
Figure 1.2 Microservices architecture design 8
Figure 2.1 Service choreography 16
Figure 2.2 Service orchestration 16
Figure 2.3 SOA Scaling 18
Figure 3.1 Proposed Enterprise Software Architecture Design 22
Figure 3.2 Request lifecycle in API Gateway 26
Figure 3.3 A sample request message JSON 27
Figure 3.4 A sample response message JSON 27
Figure 3.5 Private queue usage for inter-microservices communication 30
Figure 3.6 [44] Client-side service discovery 32
Figure 3.7 [44] Server-side service discovery 33
Figure 3.8 Flow of the message director 34
Figure 3.9 Providing JWT token 38
Figure 3.10 Authentication and authorization flow 39
Figure 4.1 RabbitMQ and HTTP RestAPI performance comparison 43
Figure 4.2 Bubble Sort Response time for an integer array of 10000 items while instance count increase 44
Figure 4.3 Message processing velocity for Figure 4.2 test case 45
Figure 4.4 Performance comparison of monolith and proposed microservices for database bounded operations 46
Figure 4.5 Comparison of error rates percentages of monolith and proposed microservices for database bounded operations 47

LIST OF TABLES

Table 4-1 Server Dedication Demonstration 41
Table 4-2 CPU Usage Percentage According to Concurrent Thread Count..... 42

1. INTRODUCTION

The passing years in software engineering forces us to find better ways for developing and deploying software applications. Every company in this sector has been developing with the help of lessons learned, as well as observing new generation technology companies to gain favor from their useful and successful approaches. The spread of technology usage provides opportunities, which can be captured rarely and of which millions of people seek. This kind of chances may have possibility to provide up to billions of dollars profit to the companies, which could take the opportunity. Often emerging trends quickly vanishes. In general, the hardware and software architectural limitations block old fashion, big and indolently evolving companies to catch the trend of change. By the pressure of the new technologies wind, the vast majority of the companies need to change the way they produce software systems by departing from their safe zone to the more risky or unpracticed styles for shortening time to market progress.

By destroying the wall of resistance to change, agile software development aspects Cloud computing and DevOps gather increasing attention in recent years. These terms come together and Continuous Software Engineering (CSE) approach arises, CSE is defined in [1] as constructing an automated pipeline which permits aggressive increasing of the frequency of successful deployment in enterprise level applications, provided with proper tooling. In the last decade, CSE is dramatically changed with Web 2.0 and Software-as-a-service (SaaS) [2]. Every software-based product started to respond customers' demands by improving their software development lifecycles, increasing release count and decreasing deployment duration. In order to make development faster, release resilient software products and agile software development methodologies are declared. This approach focuses on the development itself by caring individuals and interactions, working software, customer collaboration, and responding to change [3].

Google, Amazon and Netflix are pioneers and influencers that define the trending software development methods and styles are followed by many startups or big scale companies. Whereas these types of striking companies evolve the software sector, the

adoption phase required to be proactive to catch the pioneers up. With increasing attention to develop fast and not to fall behind by missing the chances and the pressure of tight deadlines, as it can be guessed, the developers started to implement and deploy more error-prone applications and waste time by maintaining broken application or refactoring. Meanwhile, continuous testing is defined as continuously running asynchronously regression tests in the background of the development device to provide fast feedback regarding the last status of the code test results [4]. Even if the power of processing is increasing constantly, the running time of the tests still cause excessive Central Processing Unit (CPU) load. When these tests are run too often, the developers have to wait longer for the results of the tests. However, the general developer behavior is to see the confidence of the code as soon as possible after he or she developed or edited the code. If this cannot be provided, they start to avoid running tests once they trust the code they write. Unexpectedly, most of the innocent and little changes bring about the production system outage [4]. Therefore, a rapid way of running tests should be found while keeping the developers unmolested. The impression of the tests emphasizes and defines a model that allows to reduce 10-15 % of wasted development time up to 31-82% with the determination of the frequency of the tests and 4-41% by changing the test order [5]. The introduced model runs test asynchronously on the development environment without requiring explicit trigger. The immediate notifications regarding the safety of code, lead the developers to have an opportunity to discover the errors and correct them while their focus is on the task and their knowledge of business and code is still fresh.

1.1. Developer Operations (DevOps)

The market expectations are changing continuously. IT infrastructure operations and management styles are also rapidly transforming to settle proper operational ground. The customers are reluctant to wait even six months for a comprehensive feature set or a considerable release. Nevertheless, the software development enterprises have difficulties to align their operations with the market needs and they call for continuous testing. Agility and lean methodologies are two main solutions adapted to respond the market demands. DevOps is the idea that emerged as the combination of these two approaches [6]. DevOps is defined and suggested for organizations to

optimize the Software Development Lifecycle (SDLC) as the five continuous practices including continuous planning, continuous testing, continuous integration and continuous deployment [7]. In a sense, DevOps can be thought as an agile infrastructure where all the development and operations are managed in an agile service-oriented way.

DevOps is the motivation or perspective which influence the way how you shape your development teams, organize the parts of the systems and build them. In another words DevOps is a set of practices which are applied to minimize time cost from code editing till the deployment of the code to production while ensuring the intended results are satisfied with high quality [8]. Agile methods have contributed to the performance and producibility of the software development teams. Although cross functional teams are established, there are some other teams in SDLC like stakeholders, deployment team, monitoring and maintenance team. DevOps aims strong and continuous communication with predefined rules [9] to promote a culture for shortening the loop of implementation, deployment, operation and feedback [10].

DevOps is split into three primary practices as infrastructure automation, continuous delivery and site reliability engineering [6].

Infrastructure Automation. Design of the infrastructure and OS which on is required by the applications to be deployed and ran.

Continuous Delivery. The automated delivery steps come after the implementation of code like building the solutions, running automated tests and deployment of the ensured application.

Site Reliability Engineering. Monitoring the operations of the working systems and taking proactive action to keep the systems always up [6].

1.2. Cloud Computing

In compliance with technology market improvement, people's expectations are increasing. The tolerant of any kind of delay has never been low and irremissible among technology consumers. For most of the software providers, provisioning fluctuating workload is a big challenge to offer end-users guaranteed Quality of Services (QoS) by adapting existing on-premise IT infrastructure. There are many definitions of cloud computing on different resource. But it can be defined as the distributed computing technology inherited from on-premise infrastructure with providing many new feature

set. Cloud computing facilitates a new generation, more efficient and distributed abstracted IT environment for the customers to serve higher throughput, increased availability [11].

This service-oriented grid computing concept has three main properties. The first one is abstraction which is achieved by virtualization of the underlying architecture. Abstraction of the infrastructure helps to the system admin by degrading learning required complex hardware detail or Operating System (OS) level knowledge to just a few clicks and defining some attributes of the system. The second one is accessibility which is a big challenge that must be guaranteed by IT staffs in a predictable way. Scalability is the last property which allows to increase the needed resources on-demand to ensure QoS. Then release the resources when those lack [12, 13].

Architecture. Cloud-based IT resources can be divided into three services as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [11, 12, 13, 14]

Infrastructure-as-a-Service (IaaS) is a service delivering a configurable virtual server infrastructure with the guaranty of availability and an elastic usage-based pricing model. IaaS also supplies configurations to allocate flexible resource capacity like computing unit, memory, storage and network. This facilitates to handle the highly dynamic workload to serve an application for large number of consumers. The IaaS leads its customers to achieve shortening time to market and steady end-user performance experience. The IaaS is also called Hardware-as-a-Service (HaaS).

Platform-as-a-service (PaaS) supplies an environment with development tools to developers to operate all phases of SDLC. The PaaS users can easy access to the tools over application programming interfaces (API) or web portal which is enriched with dashboard, subscription and consumption details.

SaaS is a platform where web applications or web services are hosted to serve simultaneously software solutions to end-users over the internet. One of the main benefit of the SaaS is the reduction of total cost of ownership of the applications. You do not have to buy the applications and there is no need to install them on the physical servers. Instead, subscription is the only operation to the SaaS web application or web services.

Economy. The advantage of consuming a SaaS solution is to follow the charged-by-use model. In this way, you do not have to pay for the hardware which does not commonly utilize most of the resources in most of its lifetime and there is no need to afford the cost of hardware. In addition to this, the provider of the as-a-Service handles all the maintenance operation. By this way, the service consumer disburdens the cost of IT staff employment.

User experience. User experience is the key point of all the software-based platforms. The service providers improve their facilities and ease use of the platforms on well-designed web applications. The cloud computing consumers do not want to spend much time to learn how to use. So, the demanded service must easily be customized by selecting OS and setting the RAM, disk size, CPU, network bandwidth, the application what they need. Cloud platform should meet immediate customer scaling expectations. When the demand of an application spikes or reduces, the provider cloud computing customer could reflect these changes automatically just after ordering the configured resources. Reliability is another concern that is directly affected by the user experience. Any kind of the cloud services should always be accessible for all the web and mobile clients. To achieve that the redundancy and fault tolerance mechanisms are employed.

Security. In today's world, security is the primary concern in every environment. There are two types of adoption of cloud services as private and public. Public cloud model is a multi-tenant platform and open to public via internet. The security is being endeavored with login credentials. As a more secure environment, the private cloud platform is a commodity or entity designed for using by a single tenant. The improved security is supported with logical isolation technics and some encryption methods to store business or person specific data [11, 12, 13, 14, 15].

1.3. Software Architectures

If you are working on a software product development, your software architecture choice defines how you respond to development of time pressure, tight deadlines, and adoption of the change request. It is not easy to build a cure-all software architecture. The initiatives, which settle the correct SDLC processes on a correct environment

supported with efficient tools, lead to deliver innovative features of software products in the competitive market. The successors and innovative companies must be facilitated with a well-designed, strong, resilient and agile software architecture and platform which centralizes core architectural features and makes easy to develop software products focusing barely business requirements development itself. In this chapter, we describe the Monolith and Microservices architectures which are two commonly used architecture and mention about pros and cons.

1.3.1. Monolith

Traditional enterprise-level software systems are commonly designed as monoliths—all-in-one, all-or-nothing [16]. In [17] monolith is defined as "a software application whose modules cannot be executed independently." This architecture design makes it difficult to scale and maintain due to its complexity. Besides these drawbacks, changing or fixing parts of a such system takes a lot of time during construction.

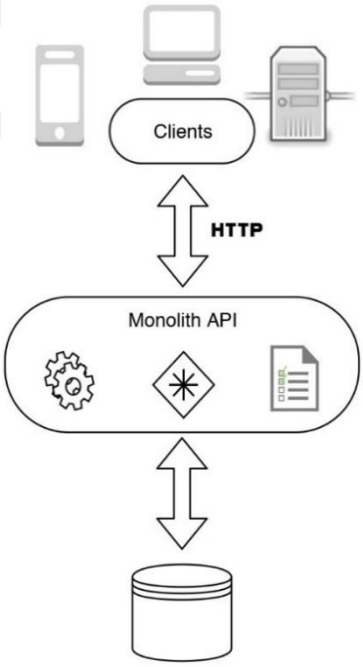


Figure 1.1 A standard monolith architecture design

It requires to spend much time trying to figure out what the monolith does. To achieve this, firstly, developers must elicit a dependency schema to understand existing dependency graph. Otherwise, adding a new dependency or updating libraries may push

the system to an inconsistent state. Due to the fragility of the system, even a small mistake would cause unintended and unknown side effects. As a solution, reasoned and well-designed regression tests should be developed to minimize the unpredicted effects of changes and keep the system up by checking the stability of all components; however, this is not easy.

1.3.2. Microservices

As Object-Oriented paradigm has grown common acceptance, abstraction of the code block and their business-oriented functionalities have started to be provided by services. This approach encourages the adoption of Service Oriented Architecture (SOA) which serves some business-specific functionalities via an interface. Application of SOA in Enterprise level systems are followed by Domain Driven Design (DDD) approach [16]. As OO architecture and DDD were promoted by Single Responsibility Principle (SRP) [18], microservice alike architectures emerged.

Microservices architecture (MSA) is a brand-new approach which separates domain-specific applications into smaller deployable services to facilitate continuous integration, scalability and reliability. Fowler, M. and Lewis, J. define the MSA style as an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [19]. In [20], MSA is defined as a composition of small services, each running in its own process and communicating through lightweight mechanisms (often over REST APIs). MSA is built around domain-specific business functionalities and employs a full-stack implementation of software for its business area [21]. Each of the services can be run independently on its own environment by connecting to a lightweight inter-service communications infrastructure [22].

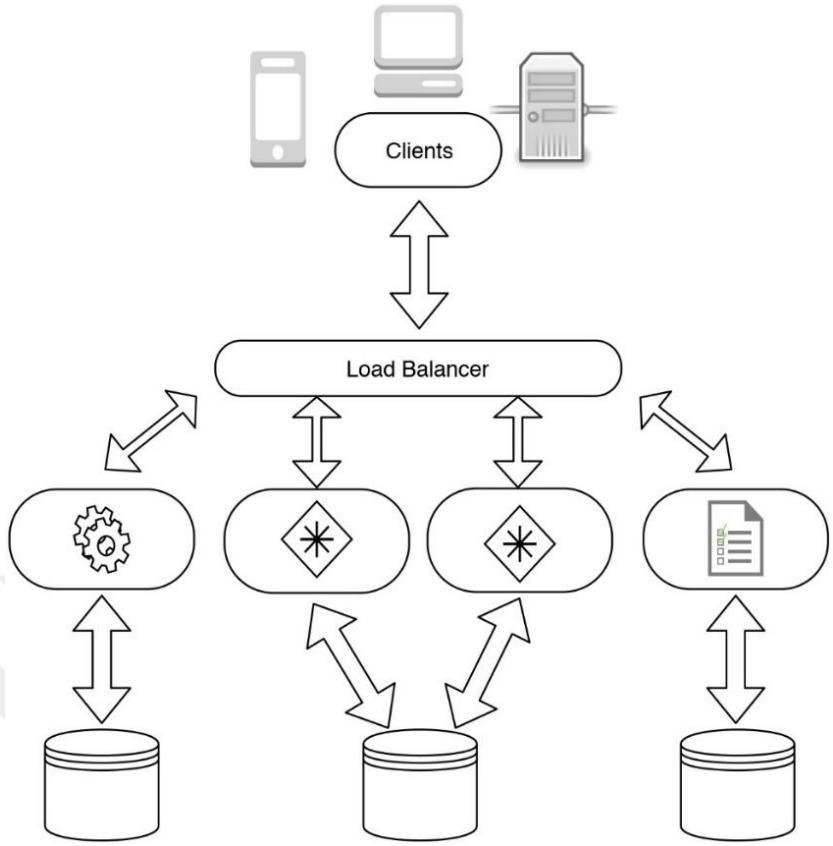


Figure 1.2 Microservices architecture design

1.4. Related works and motivation

In the literature, there exists a variety of research studies in the context of micro-services architectures. (Aderaldo and friends, 2017) [23] focus on selecting a community-owned architecture benchmark to support repeatable microservices research. Takanori et al. [24] analyze the behavior of two versions of the benchmark, microservice and monolithic. In [25], Amaral et al. aim to compare the CPU usage and bandwidth utilization benchmarks in the monolithic architectures where the whole system runs inside a single container, or inside a microservices architecture where one or few processes run inside the containers. Hence, the two models of microservices architecture provide a benchmark analysis guidance for system designers. In [26, 27], authors define the steps to construct a microservice-based service software. However, the defined steps include only some high-level suggestions for the determination phase of the general software microservice layers without giving detailed information about

building the entire design. In [16] and [28], Boner discusses strategies and techniques to build scalable and resilient microservices and design the communication model. In general, while studies focus on MSA, they do not dig into the inner detail of architecture [28, 29, 30]. The given examples do not provide enough information and comparison to develop the right solution in MSA point of view. In [31], the researchers define a monolith and a microservices web application systems. Then they just compare the cost of the monolith and MSA from the development and deployment perspectives.

As a first step towards filling this gap, this study proposes, compares, discusses and illustrates the use of proper architecture, protocol, data format, web server and deployment methods for a green-field project implementation of an enterprise-level application with the following design considerations:

- The system needs to be scalable. The system should be able to grow horizontally up to 50 times of its initial load.
- The system needs to be highly available. There should be no single points of failure. The required uptime is about 99.5%.
- The system must be maintainable in the following sense: The impact of any change to the system must be easily predictable and reversible; as such, risks should be foreseeable and containable.
- The system should be resilient to failures in the following sense: Any failures should not cause unspecified operations, and the business state of the system should always remain consistent.
- The response time to a user request is constrained to be less than 600 ms time to first byte (TTFB).
- Business processes should be easy to implement, modify, route, measure and report.
- The whole system should be monitored for interactions, transaction times, and errors.

For the proposed architecture of this design problem, various concepts, methodologies and patterns like MSA, queue-based messaging, Representative State

Transfer (REST) services, message formats and conversion methodologies, data persistence systems (RDBMS, NoSQL) will be analyzed, compared and considered.

In this thesis, we suggest and evaluate a flexible software design on enterprise level that allows companies to come to the fore. We provide a thorough comparison of a microservices-based design and traditional monolith architecture.

In this chapter, basic definitions and prominent properties of the monolith and microservices architectures are defined by referring pros and cons of approaches. In addition, related works and the motivation of the thesis are identified briefly.

In Chapter 2, The Architectural evaluation of microservices are introduced with monolithic and service-based architecture. Moreover, the differences and similarities between service-oriented architecture and microservices, and monolith and microservices are shown. The purpose is to compare these architectures to understand the reasons why one needs microservices.

In Chapter 3, sub-components of microservices are shared. The aim of is to figure out the challenges of enterprise level microservices software architecture design and find possible solutions to them. Communication, service discovery, modularity, security and database selection are presented to show the components of the design and how they could be problematic in the process.

In Chapter 4, experimental results are introduced to clarify the test environment and to show performance of the implemented application of proposed architecture. The database performance is shown to emphasize the effectiveness of the proposed microservices-based architecture.

In Chapter 5, experimental results are summarized with the gained experience and research topics that address the open points for improvement.

2. METHODOLOGICAL CONSIDERATION

Creating of a software product is always risky. Unfortunately, there is no a fitting solution for all cases. Each initiative tries to create a product to figure out a problem or run and orchestrate complex operation. Hence, while you are building an enterprise level application from greenfield, you must assess the alternative approaches of a sub-challenge from different aspects. In this chapter, the possible tools, approaches and protocols are reviewed in order to design an enterprise-level software architecture which has the properties mentioned in previous chapter.

2.1. Architectural Evaluation

Fugitive market trends and the technologies dominate IT systems. The evolution of the digital ecosystems forces them to react to ever-changing context of the business models. Adaptation is the only way for technology companies to survive [29]. The enterprise architects work on transforming their enterprise architecture (EA) to hold their companies on to the life over shear surface.

2.1.1. Monolithic architecture

The simplest form of the architecture runs all the bundled functionalities on a single layer. Essentially, the monolith approach is the style of development applications in this way. The simplicity which comes from the form of the single unit application, conforms many small startup teams, then they build self-contained software applications. In most cases, the components or services of the monolith are combined and linked as a unified solution [32, 33]. However, the traditional EA has essentially three different layers. The presentation layer provides an interface so called frontend to the client. The business logic layer contains workflows to drive the procedural logic for business purposes. The last one is data access layer which abstracts the database from the upper layers serving the data access and control abilities. This segregation somehow aims separation of concerns (SoC) principle to work on parts of the monolith architecture [34, 35].

The monolithic architecture eases doing business when the scale or complexity is out of context. It has been a well-known structure for a long time. Therefore, there are many tools and applications which can ease the development. Besides its convenience, the core of application runs in a single directory. This allows developers to release easy and newly implemented version at once. In addition to this, all software infrastructural operations such as authorization, logging, exception handling, and rate limiting are integrated in a single code base, which require less effort to implement. Its performance is better when it is compared with the service-based alternatives because the monolithic application is being run on the same host and memory. By this way, the communication overhead between components which determines the response time is kept at minimum. The scalability concern can be handled in a simple way by running multiple instances of the monolith application behind a load balancer [34].

So long as the code size and complexity are relatively small, the monolithic applications work quite well. The problems arise when some feature of sets of the tightly coupled domains need to be scaled up. Over time, multiple developers can frequently develop on the same codebase concurrently. The added new features make the code more complex and establish new dependencies between the code scopes. This extreme dependency of the code blocks turns into the code spaghetti which becomes too tough to understand how the current business flows and makes harder to map relations among modules, especially for new developers who join the development team. By nature of unified architecture of the monolith, the developers could face difficulties to work independently and they require much more collaboration which decreases the efficiency and productivity.

There are lots of tools and languages to develop software applications. To extend the number of development team members effectively is possible only by hunting talented developers. Addition to the difficulty of reaching talented candidates, their knowledge and/or experience level of the programming language are other possible obstacles. If you tend to use a new language or technology, you must rewrite the whole application. The technology and language dependency might be considered as another drawback in competitive environments.

The agility of the architecture may allow to degrade the time to market by using a variety of frameworks and languages apart from the existing ones. Code merging, building, unit and regression testing and deployment may cause a considerable increase in deployment preparation and deployment time. As we publish the monolithic system as a complete, possible development or testing mistakes may increase downtimes and failure cost [1].

Scaling is another obstacle and it costs systems in which number of transaction per hour fluctuates. Since, the monolith does not have a modular structure, entire application needs to be scaled rather than mostly used parts of the application. Such a scaling process requires more hardware resources.

All these negative effects of monoliths have been catastrophic for companies. Hiring talented developers is one of the key parameters that affects the end results of projects and time-to-market. Typically, top talented developers do not prefer struggling with architecture caused problems to keep the legacy systems stable for a long time. Production environment thrashing causes low morale. This may also have high effects, from increase of turnover rate to the failure of a company [17].

Consequently, the monolithic architecture is not completely useless. Due to the complications it holds, this architecture is not proper for the model we design for mid or big level enterprises.

2.1.2. Service-based architectures

Software engineering always defies to the challenges of software development that impact the future success of digital solution providers and the created applications. In the middle of 2000s, SOA concept [36] was defined as an architectural style which supports service-orientation. Service is a self-contained reusable representation of the group of domain functions which are bundled according to the extracted data from the results of services has a well-defined interface. MSA and SOA are called as service-based architectures. By these innovations, a lot of cutting-edge technology companies started to transform their EA to the first form of SOA. The adoption of new architecture facilitated better-designed and reusable business functionality service. SOA led to implement many development tools to help service modelling and orchestration

transform and develop. After a while, the failed software architecture transformation projects demonstrated how difficult to model services, settle inter-services communications and implementation cost of SOA [37]. Then, microservices became a popular topic. With MSA, software architects started to change their mind and spend more time to create decentralized sub-domain of a product which is fully responsible for its functionalities block of that sub-domain instead of designing “not well-grained”, “too big domains” [38].

Although MSA and SOA are not the same EA, they have the characteristics of the service-based distributed design [38].

Service contracts. Each communication needs a set of rules to set the parameters of interactions and manage the flow. Service contract can be considered as an agreement which specifies the input and output data between receiver and sender sides. Extensible Markup Language (XML) and Java Object Notation (JSON) are the most used contract types. The predefined protocol principles promote the ability to use different programming languages to implement application components. In case contract updates, all the platform attendant services must be notified regarding the change.

Service Availability. The main principle of SOA stands on reliable communication environment. Each client ensures that there is a working service accessed via a specific address. This service accepts all the requests complied with the service contract and responses in a timely manner.

Service Security. Security is not only the crucial issue for services, but also inevitable for all kind of technologies. If the service is accessible to the its consumers, then the authentication and authorization policies must be employed by each service.

Transaction Management. Services are separated according to the properties of their actions or business domain. Atomicity, consistency, isolation and durability (ACID) transactions are business critical for almost each application. In a distributed environment, transaction management must be offered not to harm data consistency while the requests are being propagated between services.

2.1.3. Service-oriented architecture versus microservices

Microservices is proposed as the new and improved version of SOA. MSA improves the way SOA does right over monolith [39]. At first glance, MSA and SOA share a lot in common. Because both approaches are emerged to employ SRP. The characteristics and architectural properties of these two principles are compared.

Although, MSA is inherited from the core characteristics of SOA, there are main differences between the layers and tasks of the services. The basic principle is to divide the application into parts as small as possible. MSA embraces easy understandable and maintainable fine-grained modularization concept for improving SDLC phases. MSA aims to use shared resources as small as possible by avoiding coupling between service. But this concept increases communication complexity and leverages performance issues and difficulties in the ACID operation management. Choosing to build services coarse-grained form may address better solutions for the mentioned problem. By that way, your direction turns to SOA.

MSA endorses silo-based service structure considering sharing as little as possible resources which are encapsulated to run the domain specific function in the context of processing flow [38, 39]. This allows building a self-contained service development which degrades the overall risk of the change with loosely coupling. SOA tends to lead to share common resources as much as possible through the application level. Shared common resources feeds the increase of coupling between services while reducing the duplication of domain functionalities. From this perspective, we can prefer loosely coupled fine-grained services to enable independently implementable, deployable small modules with minimized risk of change side effects. At this point, it must be highlighted that MSA needs a well-designed mediator component to manage the inter-service-communication due to lacking the responsible part, that SOA already has, coordinates the order of calling related services in service chaining. In Figure 2.1 Service choreography and Figure 2.2 Service orchestration, two types of service calling coordination are drawn.

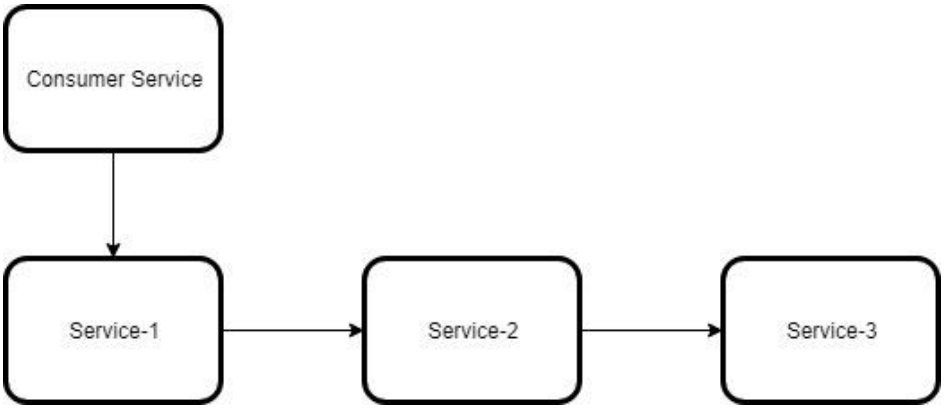


Figure 2.1 Service choreography

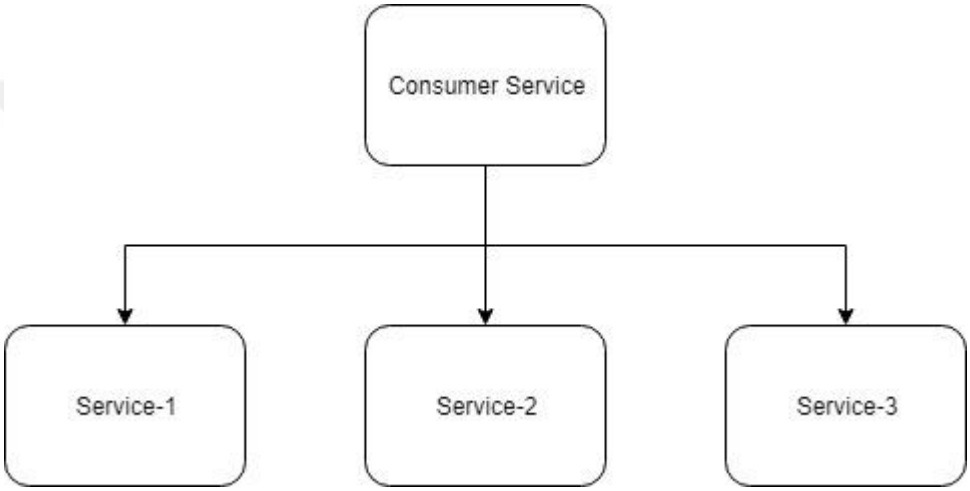


Figure 2.2 Service orchestration

The power of the microservices over SOA comes from the fine-grained, independently deployable, granular service structure. As microservices, with respect to DDD concept, leverage [39] the development of implementations that are appropriate for cloud-native applications by using lightweight containers to deploy, practicing decentralized continuous delivery and following DevOps practices in an agile way.

2.1.4. Monolith versus microservices

Microservices is proposed as opposite approach of the monolith. If the simplicity is your focus, then monolith may be the way you follow at first stage. The approach expedites simpler building and deployment. When the size of application is getting bigger, application's development team must be enlarged. After that, the complexity of

the single application increases, and the progress requires to run the parallel SDLC phases. If the scaling is an issue which must be overcome, running concurrently copies of the single large application may cause the bottleneck to handle the high-volume transactions. When the low understandability of the large code base and low quality of code problem are added to the existing ones, the monolithic approach blocks the implementation of code independently and reduces dramatically the productivity.

On the other hand, microservices is getting popular in many companies in recent years. Transformation of the software architecture leverages the opportunities of cloud computing and X-as-a-services infrastructures. MSA approach shines with its artifacts like developing and deploying independently, allowing the change the way manage the business through agile-wise path. We mention about the common motivations drive several practitioners to embark architectural transformation from monolith to microservices.

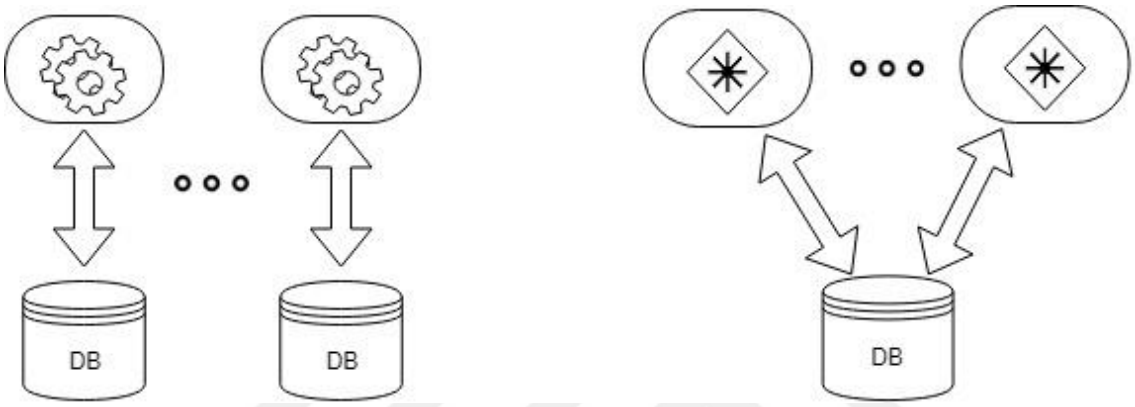
Popularity. Microservices is a cool trendy topic. Within every technical conversation, MSA is touched upon at least once with the microservices success stories of large companies. Hence, many developers and architects are curious about the detail and potential concerning microservices. They think that there are a lot of samples that are able to design and build successfully this new fashion architecture. In that case, [40, 41] several attempters of microservices confessed that the popularity of the microservices is the only reason to apply that in their companies. As the first step, proof-of-concept(PoC) is the implemented version of noncritical function proving that the applications work properly. Over time, the more popular the approach becomes, the more number of the contributors increases.

In recent years, Internet-of-things(IoT) has become another trend topic in IT world. Many of the requirements of IoT [38, 42] are addressed by microservices. The relation between those topics also contributes to spread the popularity of the microservices.

Scalability. Scalability would be one of the biggest expenditures in mid-sized or bigger companies. Running huge monolithic application entails large expenses, in case an improvement of performance for a specific function of the overall application is required. Modular and relatively small service structure of microservices allows scaling

only the expected parts of the big application requiring allocating less hardware to be executed. Figure 2.3 *SOA Scaling* shows how the monolith and microservices architectures can be scaled.

When the scaling operation is automated over an on-demand cloud platform supplier, test results show in [31] that a specifically designed auto scaled deployment mechanism of microservices can reduce the infrastructure cost up to 70% in comparison the cost of monolith scaling.



a. Monolith Scaling b. Microservices Scaling

Figure 2.3 SOA Scaling

Reusability. Each service in MSA does not execute domain or business specific operations. Some of them are responsible for running infrastructural operations like authorization, authentication, logging, monitoring, exception handling, rate limiting, load balancing. You can reuse these services not only in a single product, but also all products a company develops. This facility enables to enhance some part of the function groups easy

Container and DevOps. Container is a host, where we run the application by allocating the required resources for the application. DevOps is the set of techniques to integrate the phases of SDLC from implementation to deployment. The granularity of microservices supports the building of the proper environment to adapt DevOps principles. All the DevOps operations can easily be executed independently by different development teams for microservice. It eases the usage of container to increase resource

utilization, as well. These facilities simplify the developer's life from the aspects of deployment, monitoring, managing and recovering services [15].

Resiliency. Fault tolerance is one of the most important benefits of MSA. In case of failure of a component in monolith architecture, all the functionalities terminate, and the system is totally broken by the failed component. In contrast, MSA embraces to build isolated environment for each service. Hence, the failed part does not impact the whole system. To prove more resiliency to the system, the circuit breakers pattern [43] should be implemented and additionally, auto restarting mechanism of the failed service empowers fault tolerance.

Technology Stack. Building the system as a composition of services in microservices architecture enables the usage of variant technology stacks within each service. Breaking the system into small domain-oriented pieces allows to choose the efficient technologies for the sake of performance improvement or fast development. Different parts of the overall system may have different tooling expectations to be satisfied. For example, it would make sense to use python as programming language and document base database to store data in a service for running artificial intelligence (AI) operation. On the other part in the same system may execute transactional operation which needs to use a transaction database. Even Java or C# may be chosen as programming language to develop that service.

The hosted potential risk of using new technologies is one of the biggest barrier to adopt [44]. Within microservices, small components enable to show how new advancements of technologies enhance the current system. Thanks to the polyglot nature of microservices, you can try new programming languages or even new databases or newly released framework, as well without affecting the whole system to observe the improvements.

Time to Market. MSA [45] shortens time-to-market of developed products. Small teams can be more productive when they are working on a correspondingly small code base due to their augmented mastering on specific business domain. They can develop independently. Because MSA tries to minimize communication and coordination overhead among development teams. Furthermore, MSA enables the companies to distribute responsibilities among teams and facilitates the alignment

between software architecture and company organization. The delegation of development teams supports encouraging service ownership, which leads the teams to be organized for taking high level technical decision, causes to develop appreciated products faster.

Replaceability. Microservices can be consumed via predefines interfaces. So long as a service offers the same defined interfaces, any services can be replaced with their new versions. If you keep presenting the same interface, it does not matter what languages or technologies you use while developing the replica of the related service. In addition, MSA degrades the cost of replacement of bad-designed services and incorrect technology selection. The developers feel comfortable with reimplementation of services when the need accrues.

Maintainability. The granular structure of MSA leads a reduction in the complexity of code. If the code contains just a few hundreds of lines, it will put across the flow of business or the relations between code blocks. However, the developers can understand easily and do not hesitate to change the code when is it required. Otherwise, any maintenance or change the developers operate could cause unexpected failures.

In addition to the valuable benefits, MSA requires inevitable extra cost by opening the door of complexities in comparison to the monolith. Few of these issues are related to architecture design, like dividing too large systems into MSA style consistent sub-domains, determination of combination business capabilities that have to be served together, bounding data layer to make the microservice completely isolated, service registration and service discovery, message dispatching, event-based communication, queueing, finding the right client instance after asynchronous response fetched. The cost of MSA does not remain limited to the above-mentioned design time costs. Extra machinery, developers, tools and platforms bring extra cost so that MSA is not suggested when you have fewer than about 60 people working on your system [39].

3. PROPOSED DESIGN OF MICROSERVICES ARCHITECTURE

The goal of this thesis is to design a microservice-based architecture that targets to create a scalable system to be able to grow the system horizontally up to 50 times of its initial load. Also, the required uptime is about 99.5%, so no single points of failure is accepted. That means the system needs to be highly available. Another important point that should be emphasized is that the impact of any change to the system must be easily predictable and reversible. In another meaning any failures should not cause unspecified operations, and the business state of the system should always remain consistent. The response time to a user request is constrained to be less than 600 ms time to first byte (TTFB) and business processes should be easy to implement, modify, route, measure and report. Finally, the whole system should be monitored for any interactions, transaction times, and errors.

In this chapter, the proposed design, which is depicted in Figure 3.1 *Proposed Enterprise Software Architecture Design*, is defined by giving details from different perspectives.

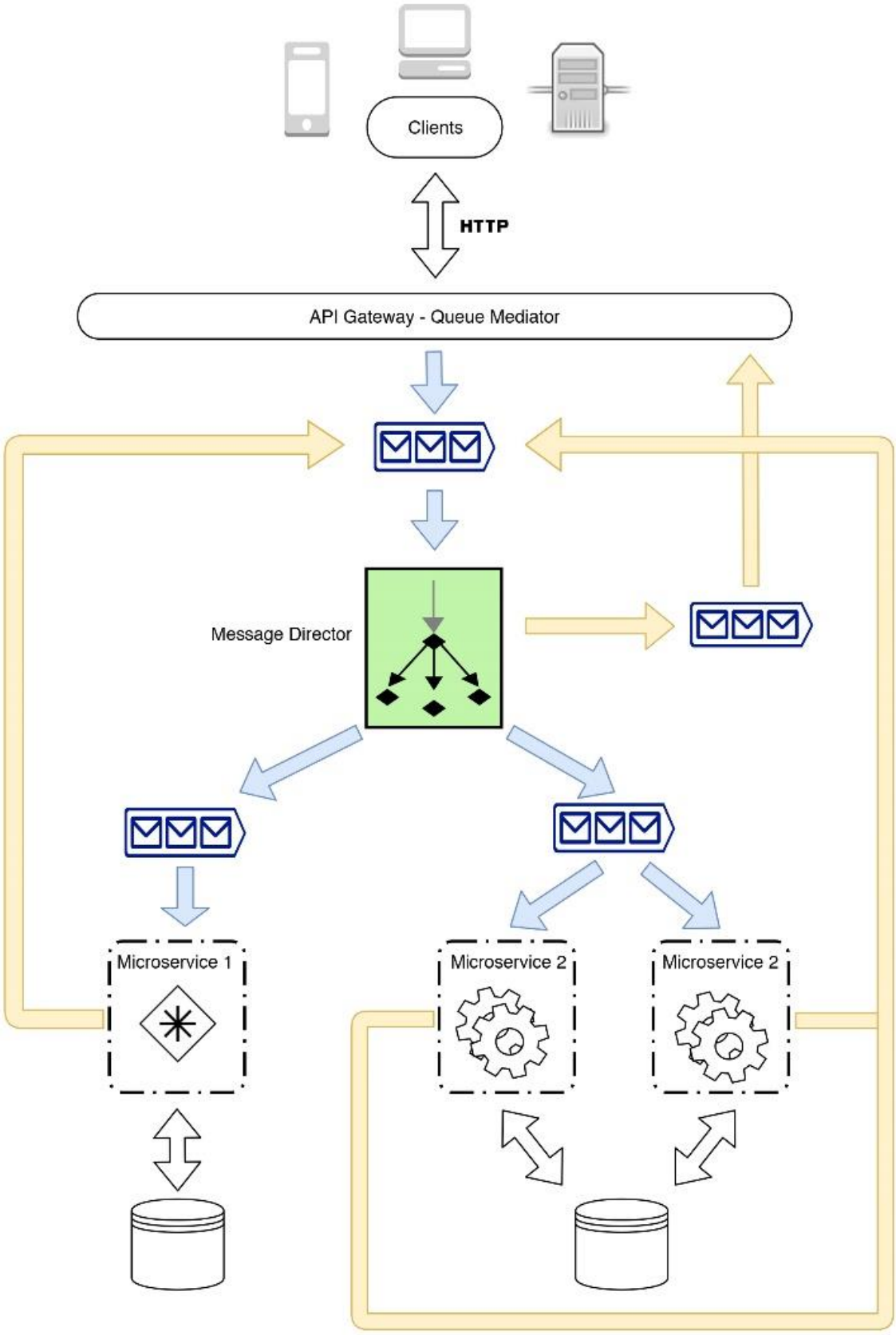


Figure 3.1 Proposed Enterprise Software Architecture Design

The outcomes and the real environment of the implemented architecture are discussed in the following chapter. However, there are some possible complication areas in the study and they are grouped as communication, service registry and discovery, modularity, security and database selection. They are explicitly discussed to resolve the possible problems.

3.1. Communication

Every organization is an alive metabolism like a human body. Like all complex metabolisms, enterprise software systems are built for making the operations of organization easy by interacting with lots of integration ends. Basically, these systems get inputs, interpret and process those values for giving them meaning and share the result of the technical operations throughout the interaction point.

Thanks to technological advancements, the speed of communication and the number of connected or data producer devices are increasing every passing day. As never before, organization is looking for the mechanism or practices to overcome whelming data transfer.

Before giving detail about the efficient possible solutions, it would be better to define the systematic communication types. The interaction types can be split into two different dimensions [46]. Each dimension invokes two options. The options for the first type of interaction are one-to-one and one-to-many transmission. These two options are distinguished by the number of the processor instance of the request. It is called one-to-one interaction, if the transferred request is process by only one service processor. If each request is processed by multiple service instance, then we call the interaction as one-to-many. The other dimension of communication contains synchronous and asynchronous options. In synchronous communication, when the client sends a request, the interaction between client and the server is blocked till the service returns the corresponding response of the incoming request. Conversely, the client is not blocked while waiting for the reply in asynchronous communication.

In one-to-one interaction, the send/receive practices can be categorized as one-way, synchronous round-trip and asynchronous round-trip [46]. The one-way model is a

practice of sending a request without expecting to get a response from the other side. This type of commination is commonly used for notifying the service provider about something. The synchronous round-trip way is applied when a service consumer needs the information from the service provider side to proceed to the next step in its business. Therefore, the client side must wait patiently during the preset reliable service timeout duration. As the last model, asynchronous round-trip is put in practice for long-running operations. The client side sends the request to the service and it is aware that the process may take relatively long time. While the server side is preparing the expected result of the request, the client-side ties up with other things until the service response is received.

The one-to-many interaction is simply the basis of publish & subscribe pattern. Basically, the publisher just publishes the message via a channel. At that moment, all the subscribers of this channel listen, and they are informed about the message. If there is at least one subscriber which is interested in the incoming message, then the message is caught and processed.

Every enterprise service system uses the combination of the mentioned practices. In MSA, there are two types of communication issues needed to be addressed. As entry point of the whole system, the API gateway is the first one. The Inter-Process Communication (IPC), which is the other one, is procured over predefined protocols. MSA offers a decoupled service which is isolated within their bounded context. In any case, synchronous communication causes to elicit coupling between services. That is why, asynchronous communication is the only way to build an MSA using isolated and decoupled services silos. Another important principle is not to share resources between services. The adoption of an MSA requires to develop the system communication on a stateless design.

3.1.1. API gateway

Each backend system must provide a common facade regardless the architecture of system to make accessibility and integration easy. There is no easy way to manage the interaction points of the enterprise systems. As enterprise architects spend a lot of

time working on development of efficient and speed APIs to proceed one step ahead from what is currently taking place.

Every service client may have different reasons to call the API services. Our ambition leads us to find a generic one-size-fit-all solution to handle all the requests. In compliance with microservices approach, we minimize the dependencies while creating the API gateway proposed enterprise application. So, the proposed system must have a common interface is called as API gateway to serve the service method to the variant client-side applications like mobile, web or other service applications.

The API gateway offers a single endpoint abstracting the actual business services to open the service clients. The API maps the business capabilities of an organization by grouping them into a granular format which the stakeholders of the organization can quickly understand. At initial step in design of an enterprise level system, the definition of interface must be done and then shared with the consumers of the system. Because the prepared API document is a kind of contract signed by the both sides of the service.

Two sides of our API gateway communicate on request/response mechanism. The request involves the parameters of the queried entities or the input data to trigger a transaction to insert or update something. The HTTP protocol is one of the most known protocols all over the world and has well-defined standards evolved to satisfy many kinds of demands. Due to nature of HTTP request/response communication is carried on synchronously. HTTP supplies such type of communication through blocking and awake style.

REST is an architectural pattern to ease web service development [47, 48]. Its popularity comes from its simplicity and its capacity to be built in HTTP features. Everyone who uses HTTP can easily use REST, as well. REST-based web services can be implemented in all programming languages which is capable to send and receive HTTP requests. Due to the REST-based framework which provides quickly development of web services, it is pervasive, and it is almost used as the default communication protocol for MSA based applications in EA world. Although, there is no reason to do that, REST is widely used in a synchronous way.

REST provides a kind of resource-oriented communication approach. The idea behind REST is to store resource to the server side and the to get, update or delete this

resource using HTTP methods. Unlike Simple Object Access Protocol (SOAP), REST does not dictate a descriptive document like Web Services Description Language (WSDL) to define the input and output parameters before calling web services. It is crucial that, REST lacks state management mechanisms. Since, all the operations must be stateless because the server side does not know anything about the state information between requests and responses. The state management should be handled on the client side.

We design API gateway as a RESTful web service to ease use of service methods by providing an interface. We aimed to keep API gateway as simple as possible in our design. For simplification purpose in development and service calling from the client-side, we introduce below restrictions for the usage of the API.

- HTTP POST is the only method our API accepts. This enables us to isolate the gateway from the business domain. Therefore, there is no need to write code in the API gateway codebase while you are developing the enterprise centric tasks.
- The first part of the URL is kept fixed. Only the last part of it can change regarding the action taken by client. We called the method name as the intent of client.
- Identification and authentication operations are handled inside API.

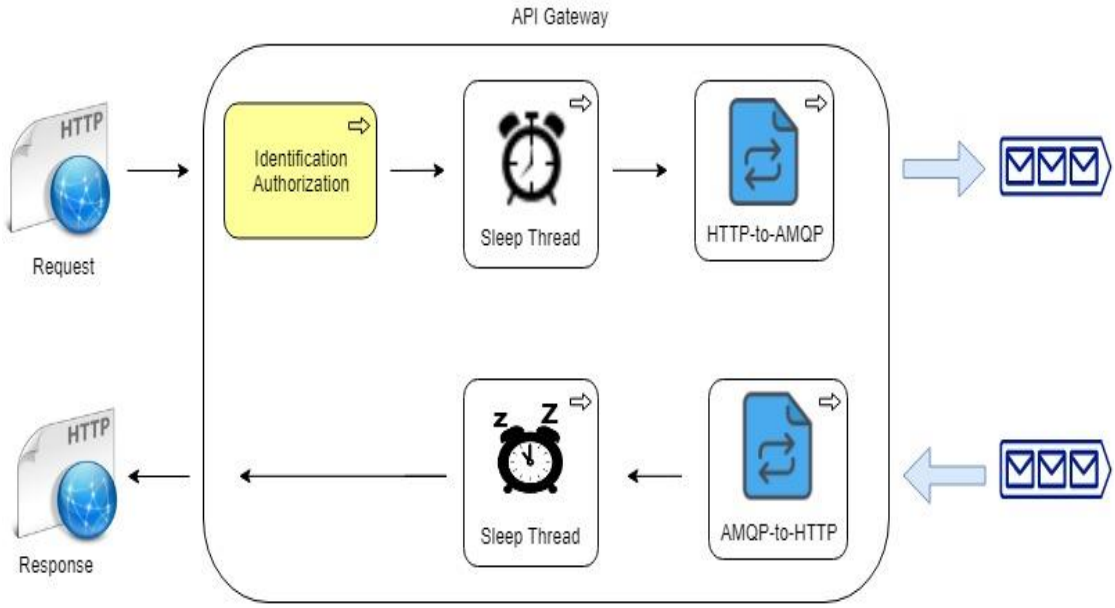


Figure 3.2 Request lifecycle in API Gateway

As it is shown in Figure 3.2 *Request lifecycle in API Gateway*, the API runs identification and authorization operation and API pushes the requester thread to a dictionary with a unique message ID to force it sleep until it receives its response via response queue while taking HTTP request timeout into account. Then proceeds transformation of the transmitted data via HTTP post operation to a predefined system message with a unique message ID and sets the message intent with posted method name. The final task of the API after invocation of receiving-event, is writing the generated JSON message to a queue. A sample request and a response JSON messages are shown in Figure 3.3 *A sample request message JSON* and Figure 3.4 *A sample response message JSON*.


```
{
    "Source": "ApiGateway",
    "Target": null,
    "Intent": "SaveDeviceLocation",
    "Tag": null,
    "JobOwner": "ApiGateway",
    "JobId": "2019-06-12-11-03-17-50645.15326",
    "Payload": {
        "DeviceId": "123456789",
        "Latitiude": 40.9739544,
        "Longitude": 29.2553255
    },
    "Username": "test",
    "Channel": "Mobile",
    "Language": "en-US"
}
```


Figure 3.3 A sample request message JSON


```
{
    "Source": "LocationMicroservice",
    "Target": "ApiGateway",
    "Intent": null,
    "Tag": null,
    "JobOwner": "ApiGateway",
    "JobId": "2019-06-12-11-03-17-50645.15326",
    "Payload": "",
    "Username": "test",
    "ResultCode": 200.0,
    "ResultMessage": "Desired return message ",
    "Channel": "Mobile",
    "Language": "en-US"
}
```


Figure 3.4 A sample response message JSON

Always business demands can change according to the market requirements or organizational provision. Each change probably contributes the increment in complexity of the architecture. The principle idea of the introduced restriction is to design the API mutable isolating from the change in backend part of the system. In a complex IT system turns to spaghetti in passing years. One of the basic responsibilities of a system architect is to make sure that the clients of an API are not exposed from the interior IT system complexity. The API should stand like a gate keeps everything related organizational situations behind.

3.1.2. Messaging Data Format Selection

In principle REST does not care what the transferred data format is. All the data formats which HTTP protocol can transmit, are allowed to transfer data in variant data formats like XML, HTML, Protocol Buffer and JSON which is the most favorite. Protocol buffer, JSON and XML can be alternatives for data-interchange format of message content transportation. Protocol buffer is the fastest one to process and the size of data with the same information is smaller than the others. But decoding the encoded data is hard without the schema. The formatted data is dense, and it cannot be called as human readable. XML is the most human readable one. Unfortunately, it contains superfluous attribute beginning and termination tags causing unnecessary increase in the size of transferred data. JSON is less verbose according to XML. It decreases the data size with removing attribute tags. Instead, brackets and curly braces are used to begin and halt a JSON component.

We prefer to use JSON, because the size of the JSON is smaller and it is more human readable. All the messages travelling throughout the system are in JSON format.

3.1.3. Inter-microservices communication

In monolith application, an interaction occurs between methods or functions. According to business flow, a method or a function could only invoke any other language level method/function. In MSA, every microservices are applications and running on their own and they must have a messaging network to communicate internal or external applications [46]. In our proposed design, we offer a synchronous RESTful-

based API gateway to manage the outer interactions. There are two options to handle IPC. The microservices can communicate over a synchronous request/response principle like our API gateway doing. Alternatively, IPC can be carried asynchronously out publish/subscribe principle like Advanced Message Queueing Protocol (AMQP).

Although HTTP is a simple, standardized, well-known and widely used protocol which supports synchronous request/response, most of the transaction in an enterprise system do not require sets of fully-synchronized operations. Just as, a well-designed asynchronous communication can pretend working as if it is synchronous. The wise-versa is not possible. In this regard, the asynchronized queue-based communications, which might be applicable, serves a reliable platform and functionalities to establish a buffered message-driven IPC between loosely-coupled microservices.

Among the alternatives like Kafka, MSMQ, ActiveMQ, we select the open source RabbitMQ [49] as the message broker since it is the most used one and it implements the AMQP. Besides, taking responsibility of load balancing with already implemented distribution algorithms, this communication type makes the system more resilient to failure by keeping messages in queues in down times of the system. Furthermore, it eases scaling by supporting publish-subscribe messaging infrastructure.

In addition to all the mentioned advantages of the queue-based message-driven communications, this method causes higher communication latency in comparison to that of HTTP. While it is easy to call a method from another component in a monolithic application, one might have difficulties implementing a system to handle calls from another microservice by distinguishing inter-service messages from common bus messages with a private queue as it is depicted in Figure 3.5 *Private queue usage for inter-microservices communication*.

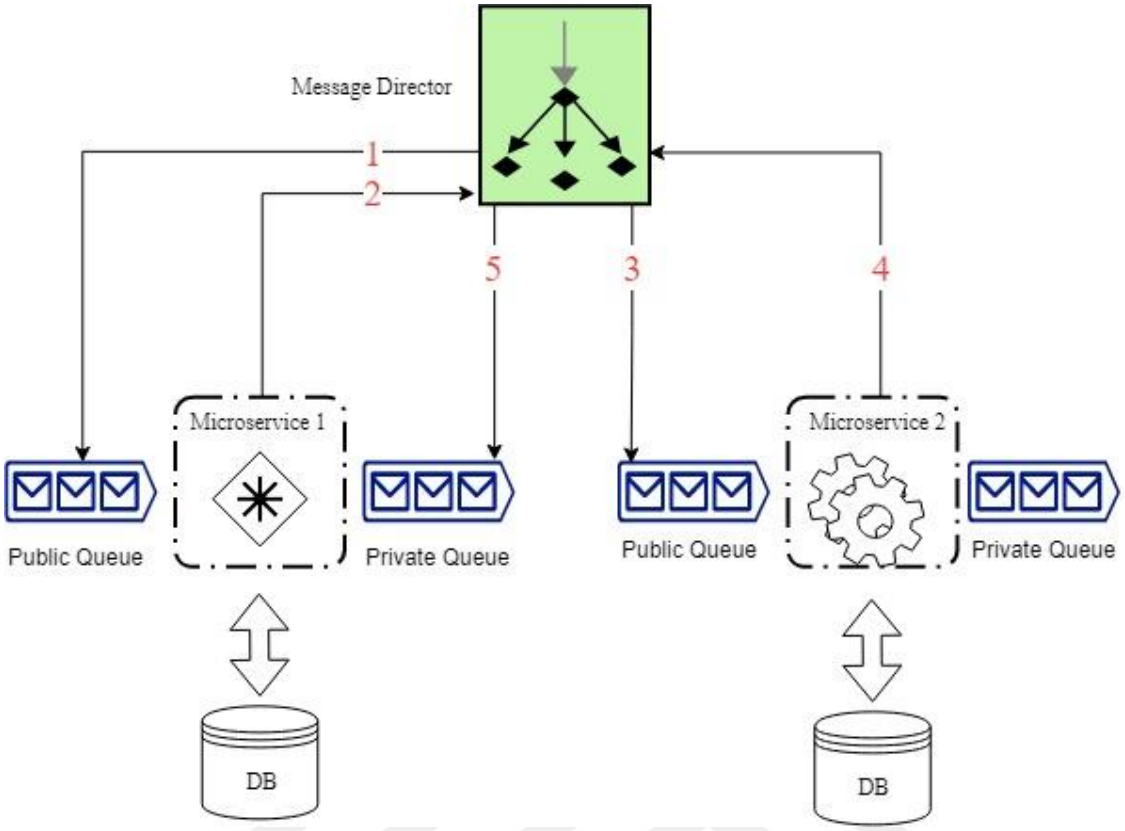


Figure 3.5 Private queue usage for inter-microservices communication

3.2. Service Registry and Discovery

Despite all the benefits in which are mentioned in this thesis microservices architecture offers, excessive challenging development tasks are accompanied by these benefits due to the dynamic nature of distributed systems [16, 28, 43, 50, 51]. While dividing the system into smaller applications can ease the management of business, addressing of dynamically scaling service instances is turning into expectedly a tremendous obstacle. Unlike SOA, maybe the most compelling challenge is service discovery in MSA. SOA often figures this problem by implementing service discovery as a task of Enterprise Service Bus (ESB).

In order to call a service, it is required to know the address of it. Additionally, the message mediator mechanism must know at a given time how many instances of services are running in which network locations. If the service location is fixed, you can store its address to somewhere like a database or an external config file to fetch it when

you need to use, or you can hardcode the address in code. Unfortunately, microservices does not present such an environment where all the involved services are able to be deployed statically to the same network locations. This approach is entirely paradoxical with the principles of MSA. Because the running instances of services are changing dynamically in case of application failure, new version upgrading, autoscaling on demand.

Service registry is the first step of service discovery operation. There must be an enterprise level common repository which is accessible by all the distributed services in MSA for storing information about the set of instances of each microservice. This repository must be kept up to date to present seamlessly service discovery. To address this, Inversion of Control (IoC) [52] pattern can be used. Simply, the pattern depends on the principle of notifications sent from each service in the systems to share information about itself. The report may contain details about which service instance it is, where it is being executed now and the definition of the interaction model. The up-to-dateness of the service registry repository has a key-value pair for the success of service discovery. Therefore, all the information about a transient service instance must be stored to repository when the instance starts up and must be deleted when it stops.

Service discovery is the mechanism which determines the current addresses of each microservice instance to the requesters by looking up regarding the requested service information in the system level service registry repository. This mechanism [21] is an obligation which is revealed from the effort of microservices to keep the services dependencies loosely coupled. Service discovery fundamentally is an ability to find all the services each other at run-time.

There are two locations where service discovery can be achieved in a big scale microservices design. Those are client or server sides.

Client-side Service Discovery. As the first option service discovery can be operated in client-side. In this circumstance, the detection of service addresses of all running service instances must be carried out by client. Aside from determination of service addresses, the implementation of load balancing algorithm is supposed to be done from client side.

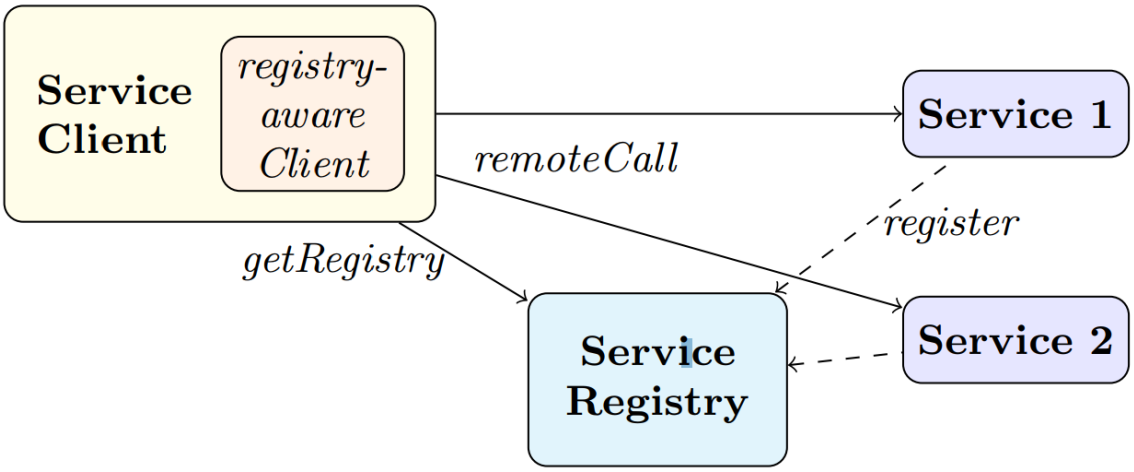


Figure 3.6 [44] Client-side service discovery

In Figure 3.6 *[44] Client-side service discovery*, client-side discovery is demonstrated. The client-side is aware of requirement that the calling information registry of a service must be looked up and fetched before service call. Since, the client-side enquiries necessary record from service registry repository. Then, it instantiates the service calling operation.

Despite this pattern requires to implement complex discovery and load balancing algorithms according to the used client language, it is straightforwardly simpler and effortless for server side. But this pattern is a bit problematic. No one wants to hand over their system at the mercy of another system [28]. Regardless of how an MSA based enterprise system is built resiliently, external systems may become dangerous by using blocking protocols and exposing riotously the system with overloaded requests which are not able to be handled. Another drawback of this pattern is to increase coupling with client-side and the service registry repository.

Server-side Service Discovery. The second pattern is to handle discovery on server-side. As it is depicted in Figure 3.7 *[44] Server-side service discovery*, the clients make requests to a static known address. The server-side router queries the corresponding service of the incoming request from the enterprise service repository and then forward the request to an available instance of the related microservices according to the service registry.

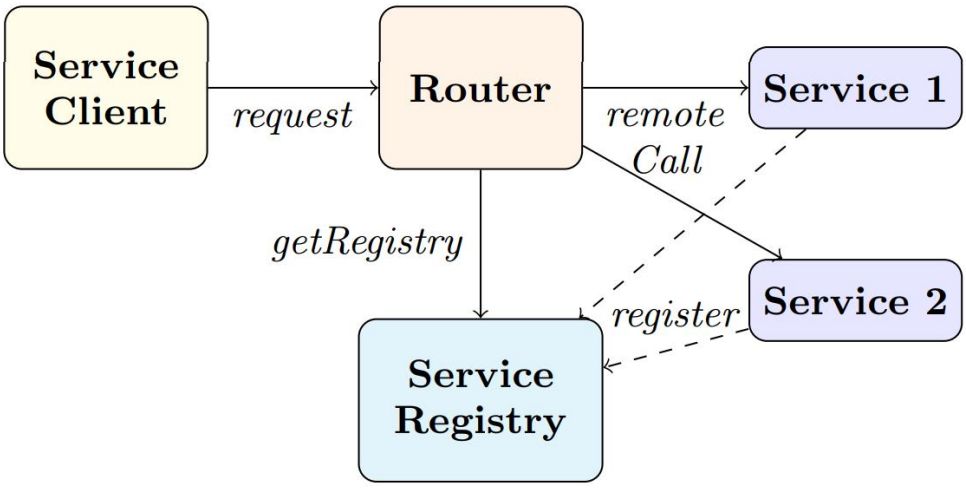


Figure 3.7 [44] Server-side service discovery

With server-side discovery, the customers are not required to know about internal services and their instances and all the system detail can be hidden from outer world. However, this pattern comes with some new challenges like high availability.

When those two approaches are compared, it can be inferred that the client-side service discovery is not appropriate for enterprise level MSA based systems. The client-side is more proper for intranet fronted and backend interactions. Consequently, because of interaction requirements of an EA with outer world services and customer in a secure and resilient way, the server-side service discovery method is implemented in the proposed EA.

Message Director. As it is shown the flow of the message director in Figure 3.8 *Flow of the message director*, we implement a message router component with the name of "Message Director". The API gateways receives and converts the requests from HTTP to AMQP protocol and then writes the messages to the message director queue. The message director is the only component that manages all queue communications at the backend. This includes forwarding all the messages to the relevant queue by checking the intent property of message. Message director must keep an intent-to-queue routing table up to date to transmit messages to the corresponding queues. We store the mapping table in RAM and if it cannot find a corresponding registry for an intent in its routing table then it asks the intent-to-queue registry to a global cache manager to achieve reliable message routing. When the registry record of an unknown intent cannot

be obtained from global cache manager, an exception is thrown to inform the client about the situation.

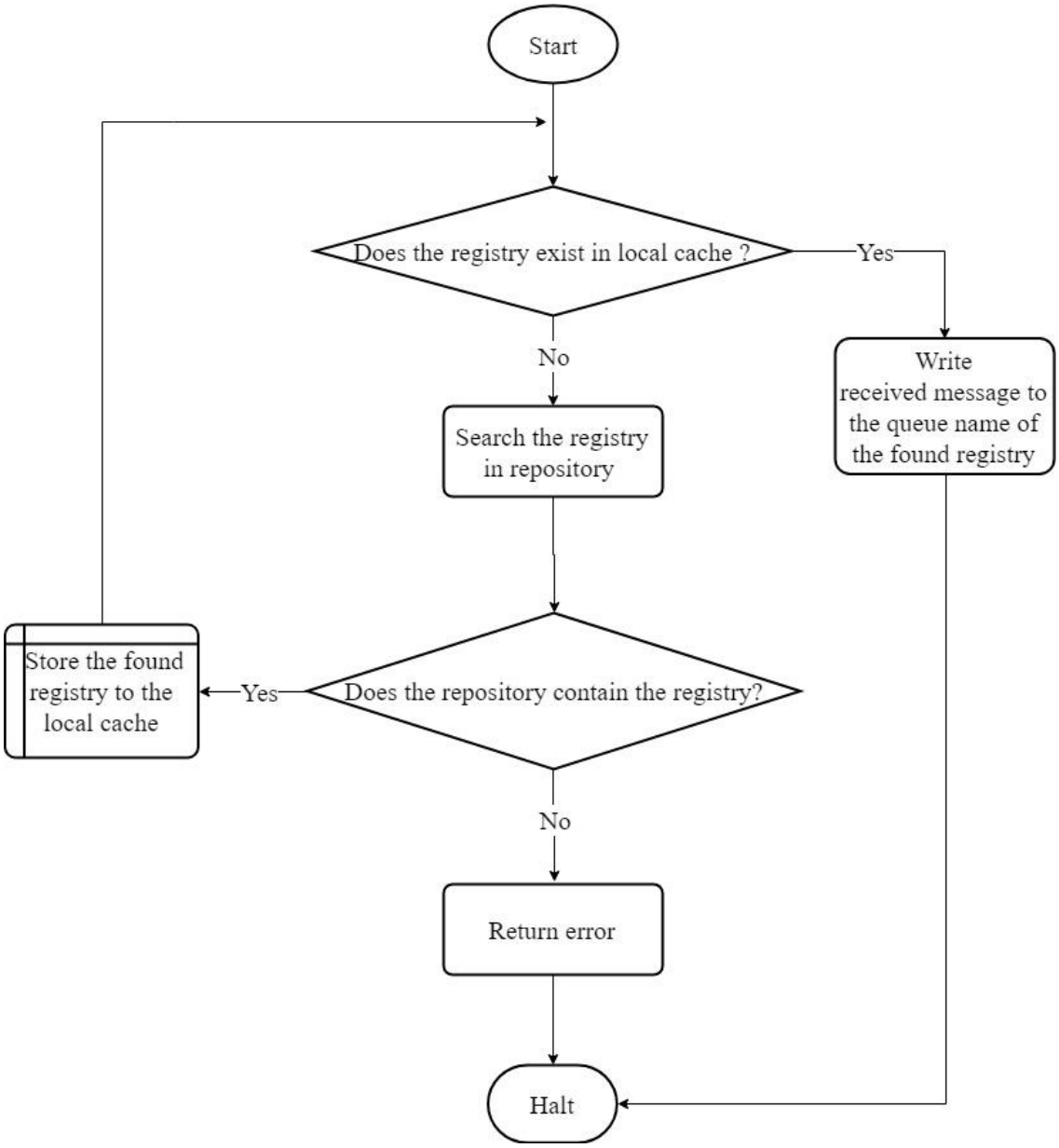


Figure 3.8 Flow of the message director

Besides, bridging the client request and corresponding microservice, it logs every request and response messages while performing the routing process. If it is intended, this module allows us to write specific rules to monitor the state of the system or to generate alerts for specific actions.

3.3. Modularity

So up to this chapter, much knowledge is shared regarding the key benefits of MSA. The basic principle lies under microservices architecture is “divide and conquer” [16] by breaking the systems into bounded subsystem contexts. The determination of boundaries for each microservices states the artifacts of using microservices. The performance of the designed system depends on how the boundaries of microservices are drawn to maximize the advantages and to avoid as much as possible downsides. [32] suggests following the boundaries of the data model to determine the boundaries of microservices. In this approach, each microservice must have a private set of tables or a private database schema or a private database which are not able to be accessed directly by other microservices.

While microservices are being modelled, loose coupling and high cohesion [44] are two key points which should be considered to maximize the upsides. These approaches identify the way which makes change of a microservice easier and faster. The decoupling can be provided throughout such a less resistant to change way. In that way, any change of a microservice should not need a change of any other microservice. The high cohesion is the other goal which is needed to be supplied with centralizing domain-centric related operations in a service. This approach also keeps related codes within a service and reduces coupling.

The microservice is a kind of small application to handle a specific task or a set of tasks in a domain and the architecture comes with mentioned coupling and cohesion problems. To be able to design a successful MSA, overwhelmingly Distributed Reactive System approach is proposed. Reactive mechanism [28, 53] is a system that focuses on asynchronous messaging for distributed architectures to help build isolated and highly collaborative services. It is a message-driven-based architectural approach [44] which composes the results of multiple calls together to run operations. The calls can be synchronous or asynchronous and the principle idea under this approach is to emit the required data from different resources and push them asynchronously when the results become ready.

In the proposed enterprise level microservices architecture design, fundamentally the reactive programming is used to decompose each request into multiple discrete

steps. Each microservice communication has been carried out over AMQP protocol while it can be written in any programming language. The microservice emits the message by subscribing to the predefined queue and extracts the intent of the message to decide the related inner method to be invoked dynamically. This domain specific applications generate proper response to each request and publish the response message to the message director queue.

Each microservice must register all the service methods that are implemented in it on the global service registry repository at its booting phase. By that way, the service discovery operation is figured out. The service registry record contains the name of the service method and the queue name that the microservice subscribes. If there is a registry record and if it requires an update, it is updated. Then, the booting microservice informs the message director module about the change to revise its intent-to-queue routing table. After service registry process, all the related client request can be forwarded to the correct microservice by message director.

3.4. Security

In IT community, security is the common concept which must be provided. Security is also a major challenge in distributed systems. The key benefits of distributed systems or microservices architecture such as granularity, easy deployment, inter-service communication result in new security gaps and specifically, small pieces of microservices architecture expand the security risk surface [17, 54, 55, 56]. Each IT system promotes a security layer according to sensitivity of its content or operations.

Providing security is a costly operation. Hence, the security level of a system can be split into sub-layers and it can be strength by keeping within the limitation of security budget. [57] proposes the hierarchical microservices security level inspiring by the OSI network layering. These layers are listed as hardware, virtualization, cloud, communication, service, orchestration. When all these sub-layers come together with the DevOps, DevSecOps term is identified. DevSecOps [58] is the philosophy which enriches DevOps with security approaches to ensure fast and safe delivery phases within an agile way.

Though, security of an organizational system has been combined as multiple security levels, protection afford may be insufficient due to threat propagation from the weakest layer to the others [59]. Standard hardware, network and OS levels precautions may not be sufficient for protection of microservices-based enterprise software architecture. However, the first three layers (hardware, virtualization, cloud) are out of concern in this work; therefore, rest three sub-layers will be observed here.

3.4.1. Authentication and Authorization

All organizational systems should identify the client of incoming request and should control its access permissions for the related resources of the request. Authentication is the identification operation whereas authorization is to check permissions of the identified client. Authentication and authorization can be provided easier in a single embodied application. However, the complexity of identification or authority is not less in microservices. Thanks to abstraction layer of our proposed design, API gateway helps to ease these security operations for microservice-based software architecture. As API gateway is the entry point of microservices-based software, it should be the first defending layer.

OAuth2 can be accepted as the standard for user authorization [60]. OAuth 2.0 is the protocol used to simplify security operations. It allows developers to process user tokens and obtains user to access a resource. The valid tokens can be used for access permission to the resource up to their expiration times.

In the proposed design, API gateway is the point where all requests are sent via HTTP and authorized with JSON Web Token (JWT) [61] to prevent unauthorized access. It is shown in Figure 3.9 *Providing JWT token*, how the token is provided before calling an API method.

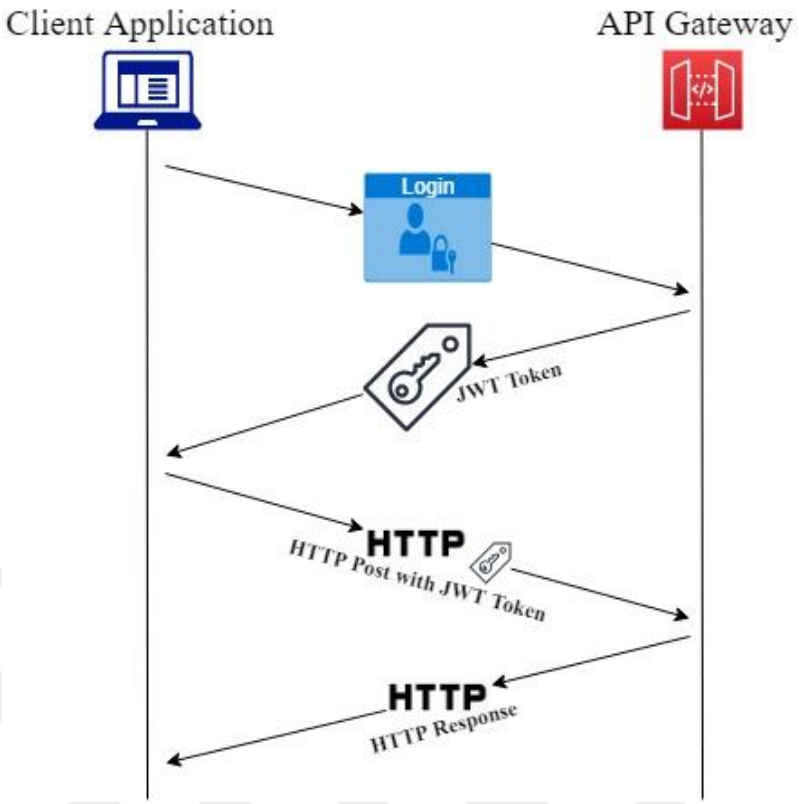


Figure 3.9 Providing JWT token

In Figure 3.10 *Authentication and authorization flow*, it is demonstrated how the authentication and the authorization can be obtained. When API gateway starts up, gets all users and theirs access rights from server side. Afterwards, all kind of authorization changes trigger data feeding to the API in order to inform API about the change of users' access rights. Each HTTP request must be posted to the API gateway with JWT token. The identity of the requester is fetched from the token and then the access right is controlled whether the user has permission to call this method.

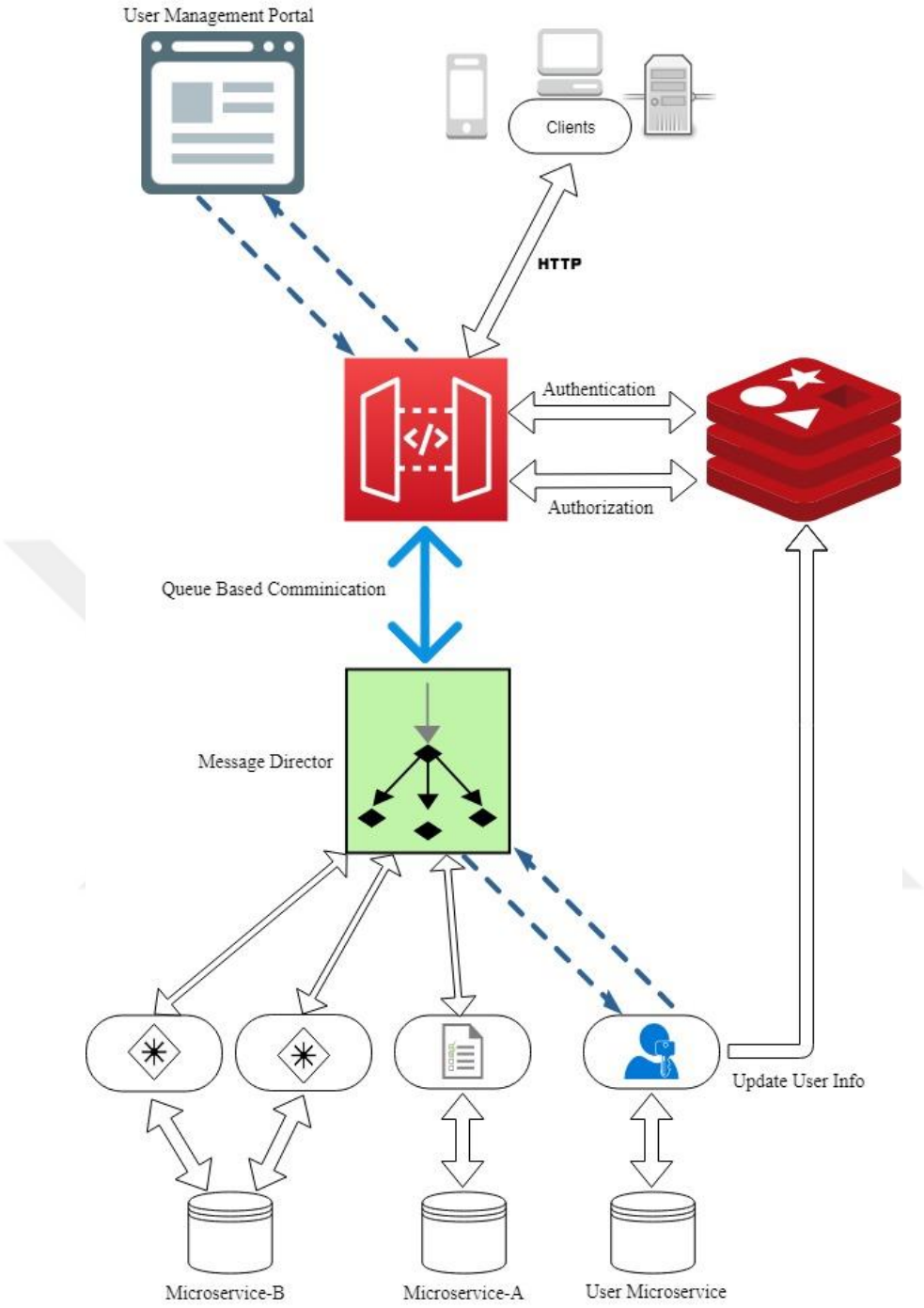


Figure 3.10 Authentication and authorization flow

3.5. Database Selection

Data persistency is one of the most important and expensive tasks for any application. One of the strengths of the microservices architecture is that each

microservice instance has an individual data model. This allows us to use a variety of database solutions such as fully supported transactional databases, open source and supported on demand databases and document-based databases in each module of the system. One can decide how to handle data persistency with regards to the planned budget, development team experience and whether business requirements must be operated transactional. We have selected an open-source transactional relational database management system (DBMS) to run operational procedures due to the project budget and NoSQL DBMS for logging.

When the cost of licensing, maintenance, official support and infrastructure are considered, selection of the open source may make sense. However, it would be essential to hire a full-time staff, who is expert in the open source database product, for running maintenance, tuning, performance monitoring, performance improvement, backup automation, disaster and recovery operations of an enterprise level production environment. The availability of staff or outsource consultancy solution has to be taken into account while making a decision on the enterprise level database product.

4. EXPERIMENTAL RESULTS

4.1. Test Environment

After we implement a prototype microservices architecture applying the selected technologies, we compare and evaluate the performance of the system. We used 11 identical virtual servers, with Windows Server 2016 OS, on the same network. 7 of these servers are used for MSA tests and others are used for the infrastructure components as shown in Table 4-1. Each equipped with Intel® Xeon® CPU E5-2680 2.40 GHz double core VCPU, 8 GB RAM and 8MB cache. The performance test is done with Apache JMeter and the test results was deduced from JMeter performance calculations. RabbitMQ server is used as message broker and Redis is used as global cache manager. IIS 8 is the web server at which the RestAPI is deployed and served.

Table 4-1 Server Dedication Demonstration

Installed Application	Server Count
RabbitMQ and Redis	1
Message Director	1
IIS Web Server	1
JMeter	1
Implemented Prototype	7

In the experiments, the results of the CPU usage percentage, the average response time, the processed message count per second are average values which are computed or observed for at least 10 minutes for ensuring to minimize the impacts of the instantly fluctuating values to increase the accuracy of the test results. The average CPU usage percentages are observed via the Microsoft Resource Manager on the conducted servers. The average response time results are computed using Apache JMeter test tool by taking average of the round-trip-times of the clients who call the analyzed services.

Apart from the RestAPI and RabbitMQ test, the tests are conducted with 100 simultaneous clients which are defined as a configuration on the Apache JMeter. In the

RestAPI and RabbitMQ performance comparison test, the client numbers vary from 100 to 300.

4.2. Performance of The Implemented Prototype Application

The first outcome of the test result is microservices architecture's having a higher network delay in comparison to monolith. As the additional latency of message director and message broker is regarded, the reason of the delay can be inferred. We measure that average round-trip time is 25ms for monolith and 30ms for our MSA prototype. The latency delta is about 5ms per request.

Each MSA instance creates a new thread for handling each received request. Therefore, the thread count management becomes even more significant for MSA instances. Firstly, we run a load test on the MSA without limiting the thread count, then we observed that the active thread count may increase up to 110. From that point, the CPU's new thread creation cost blocks the running threads to be processed in an idle or allocated CPU slot. For that reason, the instance transforms to zombie and cannot emit or reply to any request. To prevent emerging zombie MSA instances, we tried to find the optimum thread count for maximizing CPU utilization. In Table 4-2, it demonstrates how the thread count affects CPU utilization. The CPU usage percentage evaluation lead us to limit the concurrent thread count as 32. With 32 threads, CPU usage is maximized, and MSA instances are avoided from being functionless.

Table 4-2 CPU Usage Percentage According to Concurrent Thread Count

Concurrent Thread Count	CPU Usage Percentage
1	42
2	56
4	78
8	86
16	91
32	99
64	99
128	Application Fails

We test the performance of the AMQP comparing with the performance of the HTTP RestAPI. The results in Figure 4.1 demonstrate that the performance of the RestAPI which is hosted on IIS is better so long as the concurrent client number is below the simultaneous thread count limit of the IIS server. When the concurrent client count reaches to 200, then IIS starts to consume most of the time by struggling to manage the running threads. We also check the client request rate which cannot be provided a response within the 1 second timeout duration. While the error rate of the RabbitMQ is 0, the error rate for the RestAPI is 42 percentage with 300 concurrent clients.

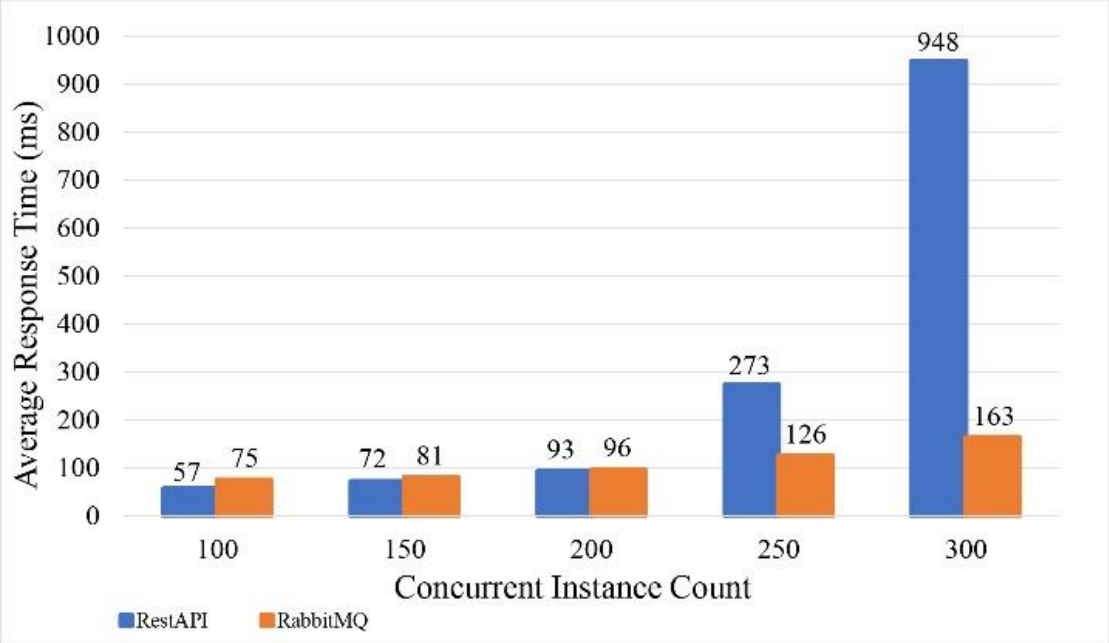


Figure 4.1 RabbitMQ and HTTP RestAPI performance comparison

The scalability performance of the proposed software architecture is observed stable and responsive in harmony with the monolith architecture performance for sorting an integer array of 10000 items with bubble sort algorithm which requires excessive CPU utilization. We generate the input integer arrays in reverse order to maximize the requirement of the CPU utilization. From the results in Figure 4.2, it is clearly seen that, when the number of concurrently running instances increase, the microservices architecture allows reduction in the response time proportionally similar to the monolith architecture.

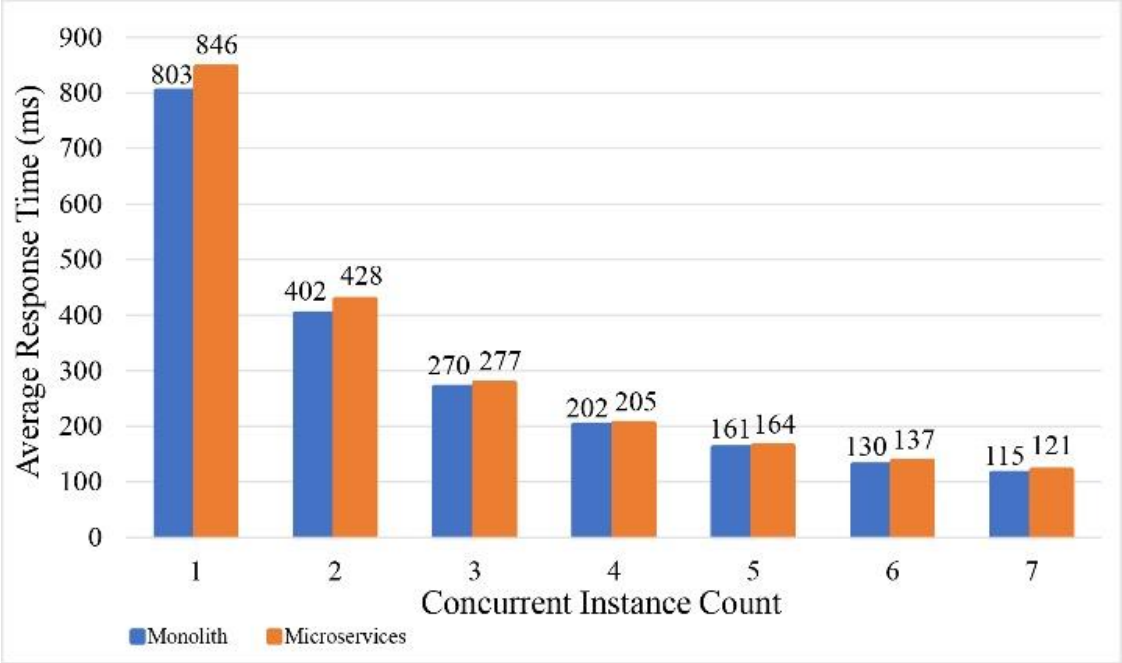


Figure 4.2 Bubble Sort Response time for an integer array of 10000 items while instance count increase

In order to foresee how we can scale our MSA, we check the message reading and forwarding count of the message director by sending messages containing a static character without expecting any response. After 10 minutes of observation, we saw that the message director can emit from message broker and forward up to 17240 messages per second. It is required to be emphasized that the message director is a point of failure module. For that reason, it is crucial that it must be run as multiple instances. This multiplication obligation also facilitates to leverage the reading and forwarding message number per second. In order to see the limits of the message handling number for round-trip operations on the proposed design, we run the prototype with single message director and an instance of microservices. Each request is replied with a single specific character to minimize the network and CPU processing latency. Under these conditions, 1120 messages are able to be replied in a second. Even if we run 7 concurrent instances on the available 7 servers, this number is multiplied by 7, we can only reach almost half of the single message director processing capacity.

We also measure number of messages handled during the execution of bubble sort operation. Figure 4.3 shows the gradual increase in the handled number of messages per second.

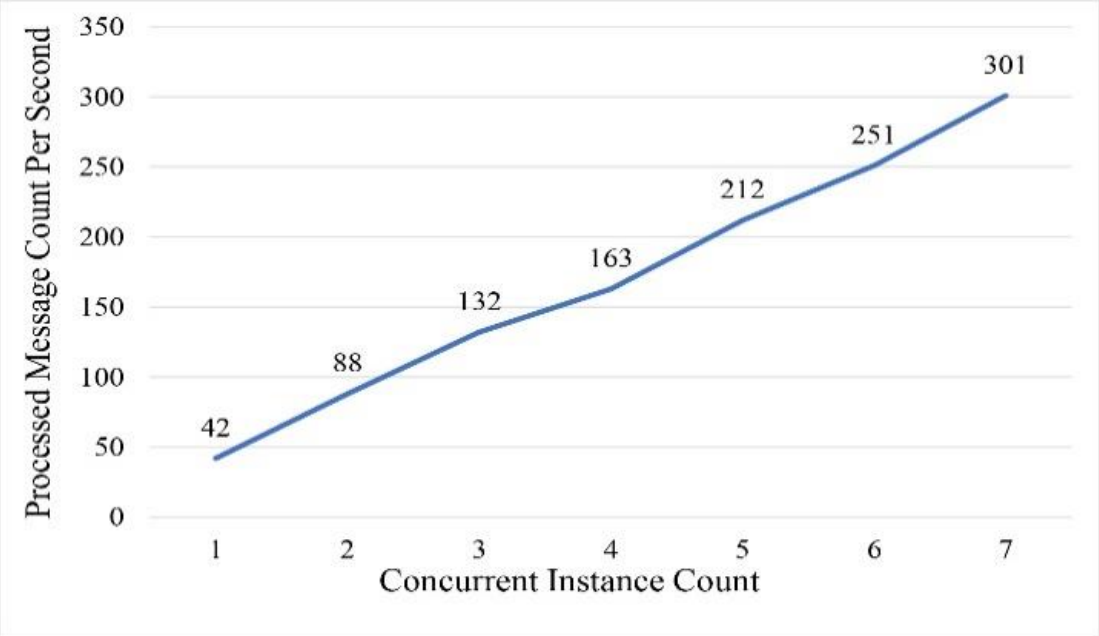


Figure 4.3 Message processing velocity for Figure 4.2 test case

4.3. Database Performance of Monolith and Proposed MSA

We compare the performance a monolith WebAPI and the prototype of our MSA design architecture for database bounded operations. In this test four different database tables are created to operate the test scenario of saving a purchasing order. For each request, three insert queries are run to save a purchasing order record, an accounting transaction and a packaging order in three different tables and an update operation is executed to update the stock record of the ordered product in the fourth table. In monolith application, the four database queries are executed consecutively. On the other hand, in microservices architecture five microservices are used. One is for the orchestration of calling four different services from four different microservices. The other four microservices are used to run the four database queries as one microservice per one query. The average response time of service responses are demonstrated in Figure 4.4.

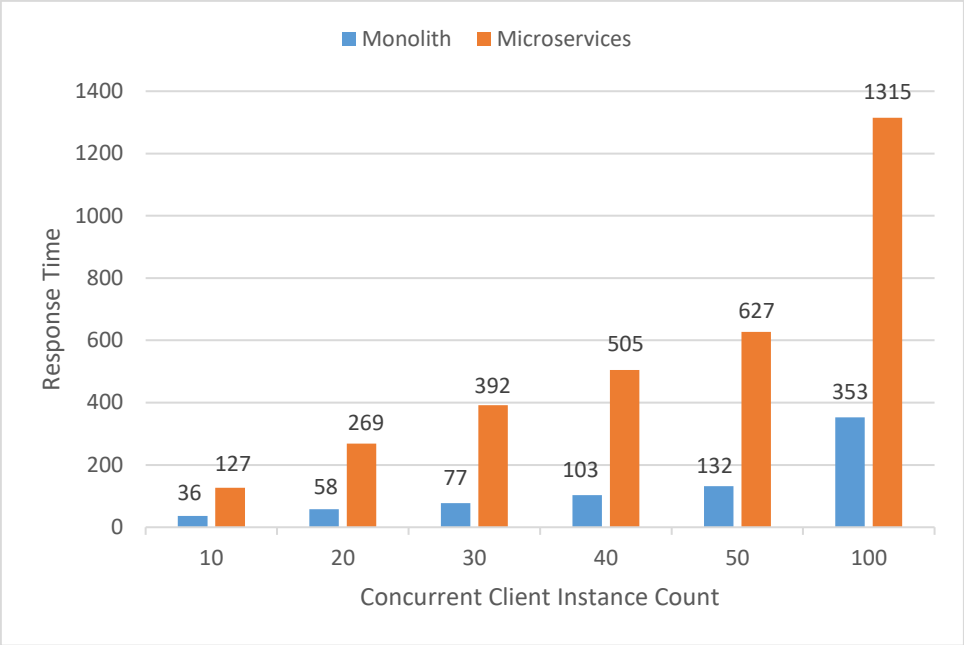


Figure 4.4 Performance comparison of monolith and proposed microservices for database bounded operations

For all the numbers of active concurrent thread, microservices architecture performs worse in comparison to monolith. The root cause of the delay in response times is the handling cost of the inter-services communication operations and orchestration of microservices. However, the error rate of replied requests is always %0 thanks to the queuing approach of our microservices architecture. Unlike microservices architecture performance, the average response time of monolithic API is less for each number of active concurrent thread.

On the IIS web server, each request is handled by a single thread. If the thread count reaches a limit, which depends on the IIS configuration and hardware resources, IIS server spends too much time for thread context switching to manage the running threads and then rejects to handle new requests. In our test, any kind of configuration is not done to restrict the concurrent thread number of IIS web server.

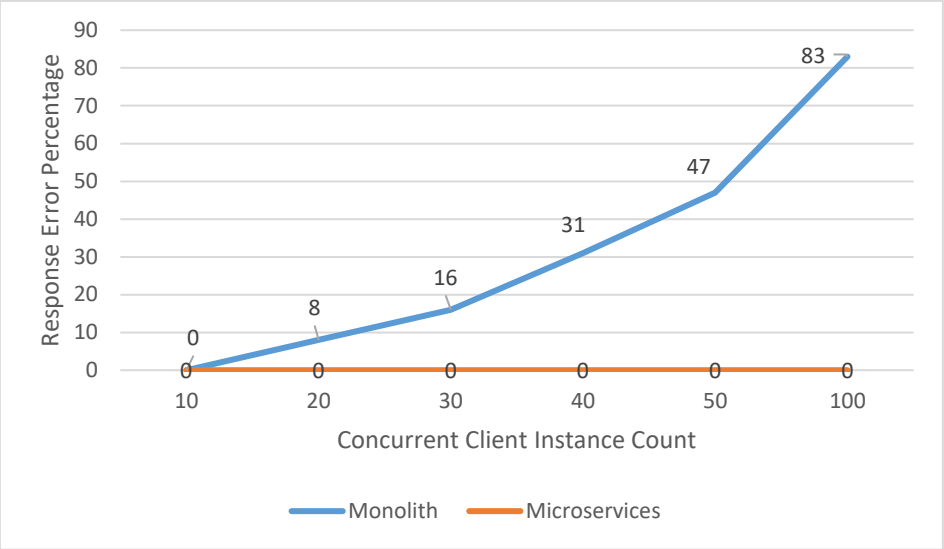


Figure 4.5 Comparison of error rates percentages of monolith and proposed microservices for database bounded operations

As it is shown in Figure 4.5, while the clients are sending new requests continuously to IIS server, the percentage of the unprocessed request is going up when the number of concurrent thread is above 10. The web server rejects to respond some of the request due to capacity overflow of thread management on IIS server.

5. CONCLUSION AND FUTURE WORK

This study presents a software architecture design which aims to satisfy the requirements such as scalability, reliability, maintainability, resilience to failures and simplicity. We implement a prototype of designed architecture based on microservices and provide performance tests for evaluating the ability of the model to satisfy demanded requirements and comparing it to the traditional monolith architecture.

The experimental results show that the proposed system provides almost similar performance compared to the monolith one. Although it causes approximately 5 ms of architectural delay, the proposed MSA system can be scaled up to tens of times compared to the initial load expectations, owing to the performance of designed message director. Thanks to the modularity of MSA, a highly available system can be served by increasing instance numbers of each component to accomplish better response times. This modularity leads us to follow separation of concern approach while developing and overcoming difficulties of building a system that is maintainable and containable.

We use queue-based communication facilitates to keep the system stable in case of failure. The queue-based communication and our proposed message director module simplify routing, monitoring current state of the system, measuring transaction times, capturing business or infrastructure errors to report.

For further studies, a rule based or even a learning tracking and monitoring tool can be designed using existing message broker to monitor availability and performance of the system. In order not to be affected by message broker crashes, a redundant monitoring tool can be built over HTTP protocol. A trained or well-designed message tracking and monitoring tool may allow taking autonomous proactive actions, in case an inconsistent state of the MSA. For the sake of increasing resiliency, Circuit-Breaker pattern can be implemented to forward messages automatically to an alternative handler when an integration point is down, until the basic service provider gets back into circulation.

REFERENCES

[1] R. V. O'Connor, P. Elger and P. M. Clarke, "Continuous software engineering—A microservices," *Wiley Software: Evolution and Processes,* pp. 1-12, 2017.

[2] J. Bosch, Continuous Software Engineering, Switzerland: Springer, 2014.

[3] M. Fowler and J. Highsmith, "The Agile Manifesto," August 2001. [Online]. Available: http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf. [Accessed September 2018].

[4] D. Saff and M. D. Ernst, "An Experimental Evaluation of Continuous Testing During Development," *ACM SIGSOFT Software Engineering Notes,* vol. 29, no. 4, pp. 76-85, 2004.

[5] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, Denver, USA, 2003.

[6] E. Mueller, "The Agile Admin," 2 August 2010. [Online]. Available: https://theagileadmin.com/what-is-devops/. [Accessed 24 January 2019].

[7] M. Virmani, "Understanding DevOps & Bridging the Gap From," in *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, Pontevedra, Spain.

[8] L. Bass, I. Weber and L. Zhu, DevOps, A Software Architect's Perspective, Massachusetts: Pearson Education, Inc., 2015.

[9] L. E. Lwakatare, P. Kuvaja and M. Oivo, "Dimensions of DevOps," in *Agile Processes in Software Engineering and Extreme Programming*, Helsinki, 2015.

[10] D. Bruneo, T. Fritz, S. Keidar-Barnerk, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, D. Raz, A. Roth, E. Salantk, I. Segallk, M. Villari, Y. Wolfsthalk and C. Woods, "CloudWave: where Adaptive Cloud Management Meets DevOps," in *2014 IEEE Symposium on Computers and Communications (ISCC)*, Funchal, Portugal, 2014.

[11] M. Rose, "Teach Target," Teach Target, 2010-2019. [Online]. Available: https://searchcloudcomputing.techtarget.com/definition/cloud-computing. [Accessed 6 June 2019].

[12] B. P. Rimal, E. Choi and I. Lumb, "A Taxonomy and Survey of Cloud Computing System," in *2009 Fifth International Joint Conference on INC, IMS and IDC*, Seoul, South Korea, 2009.

[13] R. N. Calheiros, R. Ranjan and R. Buyya, "Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments," in *2011 International Conference on Parallel Processing*, Taipei City, Taiwan, 2011.

[14] C. Gong, J. Liu, Q. Zhang, H. Chen and Z. Gong, "The Characteristics of Cloud Computing," in *2010 39th International Conference on Parallel Processing Workshops*, San Diego, USA, 2010.

[15] H. Kang, M. Le and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, Berlin, Germany, 2016.

[16] J. Bonér, Reactive Microservices Architecture Design Principles for Distributed Systems, USA: O'Reilly Media, Inc, 2016.

[17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," *Present and Ulterior Software Engineering,* pp. 195-216, 2017.

[18] E. Evans, "GOTO 2015 • DDD & Microservices: At Last, Some Boundaries!," 23 12 2005. [Online]. Available: https://www.youtube.com/watch?v=yPvef9R3k-M. [Accessed 30 4 2019].

[19] R. C. Martin, "The Clean Code Blog," 8 May 2004. [Online]. Available: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. [Accessed 8 May 2019].

[20] J. Lewis and M. Fowler, "Martin Fowler," 24 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 7 April 2018].

[21] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino and A. D. Salle, "Towards Recovering the Software Architecture of Microservice-Based Systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Gothenburg, Sweden, 2017.

[22] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Gothenburg, Sweden, 2017.

[23] C. M. Aderaldo, N. C. Mendonça, C. Pahl and P. Jamshidi, "Benchmark Requirements for Microservices Architecture Research," in *ECASE '17 Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, USA, 2017.

[24] T. Ueda, T. Nakaike and M. Ohara, "Workload Characterization for Microservices," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, USA, 2016.

[25] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar and M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, Cambridge, MA, USA, 2015.

[26] F.-J. Wang and F. Fahmi, "Constructing a Service Software with Microservices," in *2018 IEEE World Congress on Services (SERVICES)*, San Francisco, CA, USA, 2018.

[27] H. Knoche and W. Hasselbring, "Experience with Microservices for Legacy Software Modernization," in *Software Engineering and Software Management 2019*, Bonn, Gesellschaft für Informatik e.V, 2019, pp. 101-102.

[28] J. Bonér, D. Farley, R. Kuhn and M. Thompson, "The Reactive Manifesto," 16 9 2014. [Online]. Available: https://www.reactivemanifesto.org/. [Accessed 6 6 2019].

[29] J. Bonér, Reactive Microsystems The Evolution of Microservices at Scale, Sebastopol, USA: Lightbend, Inc., 2017.

[30] J. Bogner and A. Zimmermann, "Towards Integrating Microservices with Adaptable Enterprise Architecture," in *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, Vienna, Austria, 2016.

[31] Y. Yu, H. Silveira and M. Sundaram, "A microservice based reference architecture model in the context of enterprise architecture," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, Xi'an, China, 2016.

[32] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano and M. Lang, "Infrastructure Cost Comparison of Running

Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, Colombia, 2016.

[33] A. Messina, R. Rizzo, P. Storniolo and A. Urso, "A Simplified Database Pattern for the Microservice Architecture," in *DBKDA 2016, The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications*, Lisbon, Portugal, 2016.

[34] "On monoliths, service-oriented architectures and microservices," Alessandro Nadalin, 20 02 2015. [Online]. Available: https://odino.org/on-monoliths-service-oriented-architectures-and-microservices/. [Accessed 26 5 2019].

[35] A. D., "Monolith, SOA, Microservices, or Serverless?," RubyGarage, 18 4 2019. [Online]. Available: https://rubygarage.org/blog/monolith-soa-microservices-serverless. [Accessed 26 5 2019].

[36] S. Arshed, "Monolithic vs SOA vs Microservices — How to Choose Your Application Architecture," Medium Corporation, 29 11 2018. [Online]. Available: https://medium.com/@saad_66516/monolithic-vs-soa-vs-microservices-how-to-choose-your-application-architecture-1a33108d1469. [Accessed 26 5 2019].

[37] "Service-Oriented Architecture – What Is SOA?," The Open Group, [Online]. Available: http://www.opengroup.org/soa/source-book/soa/p1.htm#soa_definition. [Accessed 26 5 2019].

[38] M. Richards, Microservices vs. Service-Oriented Architecture, Sebastopol, CA: O'Reilly Media, 2016.

[39] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies,* vol. 2, no. 9, pp. 24-27, 2014.

[40] D. S. a. M. Rezai and R. Hill, "Towards an Understanding of Microservices," in *23rd International Conference on Automation & Computing*, Huddersfield, UK, 2017.

[41] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development,* vol. 32, no. 3-4, pp. 301-310, 2017.

[42] D. Taibi, V. Lenarduzzi and C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation," *IEEE Cloud Computing,* vol. 4, no. 5, pp. 22-32, 2017.

[43] 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)*, Berlin, Germany, 2016.

[44] F. Montesi and J. Weber, "Circuit Breakers, Discovery, and API Gateways in Microservices," 21 9 2016. [Online]. Available: http://arxiv.org/abs/1609.05830. [Accessed 29 5 2019].

[45] S. Newman, Building Microservices-DesignigFine-Grained Systems, Sebastopol, CA: O'Reilly Media, Inc, 2015.

[46] E. Wolff, Microservices-Flexible Software Architecture, Crawfordsville, Indiana: Pearson Education, Inc, 2017.

[47] Chris Richardson of Eventuate, Inc, "Building Microservices: Inter-Process Communication in a Microservices Architecture," 24 7 2015. [Online]. Available: https://www.nginx.com/blog/building-microservices-inter-process-communication. [Accessed 29 5 2019].

[48] D. Jacobson, "Why REST Keeps Me Up At Night," 15 5 2012. [Online]. Available: https://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15.

[Accessed 30 5 2019].

[49] M. Rouse, "REST (REpresentational State Transfer)," TechTarget, 12 2017. [Online]. Available: https://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer. [Accessed 30 5 2019].

[50] RabbitMQ, "RabbitMQ is the most widely deployed open source message broker.," Pivotal Software, 2007. [Online]. Available: https://www.rabbitmq.com/. [Accessed 31 5 2019].

[51] Chris Richardson of Eventuate, Inc., "Service Discovery in a Microservices Architecture," Nginx, 12 10 2015. [Online]. Available: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture. [Accessed 1 6 2019].

[52] J. Stubbs, W. Moreira and R. Dooley, "Distributed Systems of Microservices Using Docker and Serfnode," in *2015 7th International Workshop on Science Gateways*, Budapest, Hungary, 2015.

[53] S. Sobernig and U. Zdun, "Inversion-of-Control Layer," in *The 15th European Conference on Pattern Languages of Programs*, Irsee, Germany, 2010.

[54] R. Chandramouli, "SECURITY STRATEGIES FOR MICROSERVICES-BASED APPLICATION SYSTEMS," [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204-draft.pdf. [Accessed 6 6 2019].

[55] 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)*, Vancouver, BC, Canada, 2015.

[56] D. Yu, Y. Jin, Y. Zhang and X. Zheng, "A survey on security issues in services communication of Microservices-enabled fog applications," 14 2 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.4436. [Accessed 7 6 2019].

[57] T. Yarygina and A. H. Bagge, "Overcoming Security Challenges in Microservice Architectures," in *IEEE Symposium on Service-Oriented System Engineering (SOSE)*, Bamberg, Germany, 2018.

[58] Sumo Logis, "What is DevSecOps and How Is it Different Than DevOps?," Sumo Logis, 13 5 2019. [Online]. Available: https://www.sumologic.com/insight/devsecops-rugged-devops/. [Accessed 7 6 2019].

[59] Sumo Logic, "Improving Security in Your Microservices Architecture," Sumo Logic, 14 5 2019. [Online]. Available: https://www.sumologic.com/insight/microservices-architecture-security/. [Accessed 8 6 2019].

[60] OAuth, "OAuth 2.0," [Online]. Available: https://oauth.net/2/. [Accessed 7 6 2019].

[61] "Homepage," [Online]. Available: https://jwt.io/introduction. [Accessed 7 6 2019].

[62] A. Singleton, "The Economics of Microservices," *The IEEE Computer Society,* pp. 16-20, 2016.

RESUME

KENAN CEBECİ

Marmara University,

Computer Engineering Department, Faculty of Engineering,

Göztepe Campus, Kadiköy, Istanbul, Turkey

Phone (Cell)+90-5302115809

e-mail: kenancebeci@outlook.com

EDUCATION

Degree, Computer Engineering Karadeniz Technical University, Faculty of Engineering, Trabzon, Turkey, 2009

WORK EXPERIENCE

PUBLICATIONS

RESEARCH INTERESTS

User Provided Networks, Wireless Network, Machine Learning

FOREIGN LANGUAGES

English