# Design of an Enterprise Level Architecture Based on Microservices

*Araştırma Makalesi/Research Article*

Kenan CEBECİ, Ömer KORÇAK

Department of Computer Engineering, Marmara University, Istanbul, Turkey
kenancebeci@marun.edu.tr, omer.korcak@marmara.edu.tr,
(Geliş/Received:26.04.2019; Kabul/Accepted:21.07.2020)
DOI: 10.17671/gazibtd.558392

***Abstract*—** Building or transformation of an enterprise software system is an onerous process which requires a 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 the software world, 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 paper, 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 paper is about designing a backend API system. The client types or technologies are out of scope.

***Keywords***— microservices, software architecture, queue-based communication

# Kurumsal Ölçekte Mikroservis Tabanlı Bir Mimarinin Tasarlanması

***Özet*—** Kurumsal bir yazılım sisteminin oluşturulması veya dönüşümü, iş ihtiyaçlarının tam olarak tanımlanmasını gerektiren meşakkatli 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 almaktır. 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 seçimi üzerine yol gösterici deneysel bir araştırma bulunmamaktadır. Bu makalede, 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.

***Anahtar Kelimeler*—** mikroservis, yazılım mimarisi, kuyruk-tabanlı iletişim

## 1. INTRODUCTION

The passing years in software engineering forces us to find better ways of 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 like Google, Amazon and Netflix 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. In general, the hardware and software architectural limitations block old fashion, big and indolently evolving companies to catch the trend of change. The software-based products started to respond to customers' demands by improving their software development lifecycle (SDLC), increasing release count and decreasing deployment duration. In order to make development faster, and to release resilient software products, the agile software development methodologies emerged. Agile development approach focuses on the development itself by caring individuals and interactions, working software, customer collaboration, and responding to change [1]. Agile approaches are supported with cloud computing and DevOps principles in order to shorten time to market [2] and to serve higher throughput and increased availability [3]. With the integration of these terms, Continuous Software Engineering (CSE) approach arises. CSE is defined as constructing an automated pipeline which permits aggressive increasing of the frequency of successful deployment in enterprise-level applications, provided with proper tooling [4, 5] and proper testing [6, 7]. CSE optimizes the SDLC as the five continuous practices including continuous planning, continuous integration, continuous testing, continuous deployment and continuous monitoring [8].

To meet the customer's sectoral expectations, the successful 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 the development of business requirements itself. For this purpose, microservices architecture (MSA) is proposed which is a brand-new approach that separates domain-specific applications into smaller deployable services to facilitate continuous integration, scalability, and reliability. Each of these small services runs in its own process and communicates through lightweight mechanisms.

In this paper, we suggest and evaluate a flexible microservices-based software design on the enterprise-level that allows companies to come to the fore. We first compare the features of our design with some of the trendiest existing microservices frameworks and then provide a thorough comparison of the performance of our proposed microservices-based design and traditional monolith architecture.

### *1.1. Related Work*

In the literature, there exists a variety of research studies in the context of micro-services architectures. Aderaldo et al. [9] focus on selecting a community-owned architecture benchmark to support repeatable microservices research. Takanori et al. [10] analyze the behavior of two versions of the benchmark, microservice and monolithic. Amaral et al. [11] 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. Authors define the steps to construct microservice-based service software [12,13]. 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. Boner discusses strategies and techniques to build scalable and resilient microservices and design the communication model [14, 15]. In general, while studies focus on microservices architecture (MSA), they do not dig into the inner detail of architecture [15, 16, 17]. The given examples do not provide enough information and comparison to develop the right solution in MSA point of view. Söylemez and Tarhan mention the pros and challenges of MSA and gives alternatives to the ready tools and approaches to overcome the challenges [18]. The researchers define a monolith application and a microservices-based web application [19]. Then they just compare the cost of the monolith and MSA from the development and deployment perspectives. Yamuç and Sürme design a microservice-based satellite ground software system focusing on 12 factors and set the migration strategies from monolith to MSA [20]. Tang et al. [21] design a system architecture for the garment sector using an asynchronous communication mechanism focusing to decompose the asynchronous sectoral business operations. Pinheiro and friends compare the standard monolith and MSA from the point of enterprise architecture governance and then define the governance principles, responsibilities, and product scopes of the MSA [22]. Huang et al. [23] build an MSA and defines the common points of microservices. In this paper, they focus specifically on the load balancer optimization and they propose a dynamic scheduling algorithm based on unary linear regression. Akbulut and Perros [24] analyze the performance of MSA carrying on three different microservice design patterns. They also describe when asynchronous communication should be preferred, and which design pattern should be used to increase the hardware usage efficiency and contribution to green computing while decreasing hosting costs.

There are also numerous practical microservices frameworks such as Spring Boot team's Spring Cloud [25], Eclipse team's Vert.x [26] and Alibaba's Dubbo [27]. All these frameworks are developed to simplify the development of distributed software architecture like microservices. Those preserve ready to use components

including API gateway, load balancer, service registry and discovery, security, fault tolerance and service governance. They are all focusing on helping developers code applications and presenting their documentation to comfort the usage of their features without giving interior detail about the implementation of the components.

### *1.2. Motivation*

As a first step towards filling the gap of a detailed enterprise-level microservice architecture design documentation, 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.

The rest of the paper is organized as follows: Section 2 presents the determination of technology stack for the satisfaction of the paper motivations. Section 3 provides a design of the proposed architecture in various aspects. Section 4 exhibits the evaluation and comparison results. Section 5 concludes the paper with the gained experience and research topics that address the open points for improvement.

## 2. METHODOLOGICAL CONSIDERATION

Creating a software product is always risky. Unfortunately, there is no 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 section, the possible tools, approaches and protocols are reviewed in order to design an enterprise-level software architecture which has the properties mentioned in the previous section.

### *2.1. Architectural Evaluation*

The market trends and technologies dominate IT systems. The evolution of the digital ecosystems forces them to react to the ever-changing context of the business models. Adaptation is the only way for technology companies to survive [17]. The enterprise architects work on transforming their enterprise architecture (EA) to hold their companies on to life. To be agile and pioneer, the architecture of enterprise-level applications should support promising initiatives.

Firstly, let us introduce basic definitions and prominent properties of the monolith and MSA by referring to the pros and cons of both approaches.

#### *2.1.1. Monolithic Architecture*

Traditional enterprise-level software systems are commonly designed as monoliths—all-in-one, all-or-nothing [14]. The monolith is defined as “a software application whose modules cannot be executed independently” [28]. 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 [29, 30]. However, the traditional EA has essentially three different layers which are shown as a combined monolith API component in Figure 1. The presentation layer provides an interface of so-called frontend to the client. The business logic layer contains workflows to drive the procedural logic for business purposes. The last one is the 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 [31, 32].

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 into a single code base, which requires 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 minimum. The scalability concern can be handled simply by running multiple instances of the monolith application behind a load balancer [31].

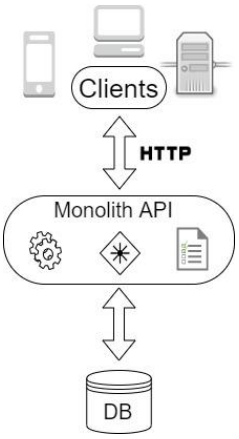



Figure 1. A standard monolith architecture design

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 the unified architecture of the monolith, the developers could face difficulties to work independently and they require much more collaboration which decreases 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. Technology and language dependencies might be considered as another drawback in competitive environments.

The agility of the architecture may allow degrading 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 [4].

Scaling is another obstacle and it costs systems in which the number of transactions per hour fluctuates. Since the monolith does not have a modular structure, the entire application needs to be scaled rather than only the 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 affects the 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 an increase of turnover rate to the failure of a company [28].

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 Architecture*

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 [33] 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 implementing 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 [34]. 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” [35].

Although MSA and SOA are not the same EA, they have the service contracts, service availability, security and transaction management characteristics of the service-based distributed design [35].

### *2.1.3. Microservices Architecture*

As Object-Oriented programming has become the dominant paradigm, an 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 is followed by Domain Driven Design (DDD) approach [14]. As OO architecture and DDD were promoted by Single Responsibility Principle (SRP) [36], microservice alike architectures emerged.

Fowler and Lewis define the MSA 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 [37]. MSA is built around domain-specific business functionalities and employs a full-stack implementation of software for its business area [38, 39]. As it is depicted in Figure 2, each of the services can be run independently on its own environment by connecting to a lightweight inter-service communications infrastructure [40].

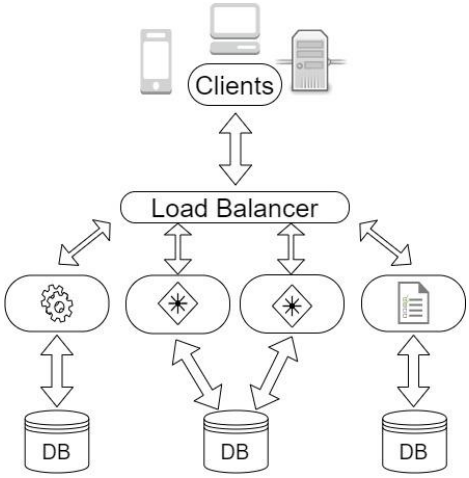



Figure 2. Microservices architecture design

*2.2. Monolith and Microservices Comparison*

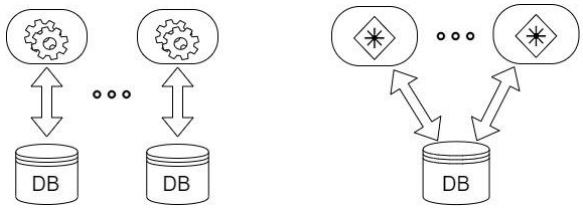
Microservices is proposed as the opposite approach of the monolith. If the simplicity is your focus, then monolith may be the way you follow at the first stage. The approach expedites simpler building and deployment. When the size of an application is getting bigger, the application's development team 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 copies of the single large application concurrently may cause the bottleneck to handle the high-volume transactions. When the low understandability of the large codebase and the low quality of code problems are added to the existing ones, the monolithic approach blocks the implementation of code independently and reduces the productivity dramatically.

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 to change the business management methodology through an agile-wise path. Now, we mention about the common motivations that 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. In that case, several attempters of microservices confessed that the popularity of the microservices is the only reason to apply that in their companies [41, 42]. In recent years, Internet-of-things (IoT) has become another trending topic in the IT world. Many of the requirements of IoT [35, 43] are addressed by microservices. The relation between those topics also contributes to spreading 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. The 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 3. SOA Scaling shows how the monolith and MSA can be scaled.

When the scaling operation is automated over an on-demand cloud platform supplier, test results show 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 [19].




a.Monolith Scaling b. Microservices Scaling

Figure 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 in all products a company develops.

**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. These facilities simplify the developer's life from the aspects of deployment, monitoring, managing, and recovering services [44].

**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 [45] should be implemented and additionally, auto restarting mechanism of the failed service empowers fault tolerance.

**Technology Stack:** The hosted potential risk of using new technologies is one of the biggest barriers to adopt [46]. 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 shortens time-to-market of developed products [47]. Small teams can be more productive when they are working on a correspondingly small code base due to their augmented mastering on the specific business domain. They can develop independently.

**Replaceability:** Microservices can be consumed via predefines interfaces. So long as a service offers the same defined interfaces, each service can be replaced with its new version.

**Maintainability:** The granular structure of MSA leads to 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 it is required. Otherwise, any maintenance or change the developers perform 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 [48].

## 3. PROPOSED DESIGN OF MICROSERVICES ARCHITECTURE

The goal of this paper 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 point 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.

To transfer technology comparison results into practice, we have implemented a prototype carrying out MSA with a message-driven development approach as shown in Figure 4.

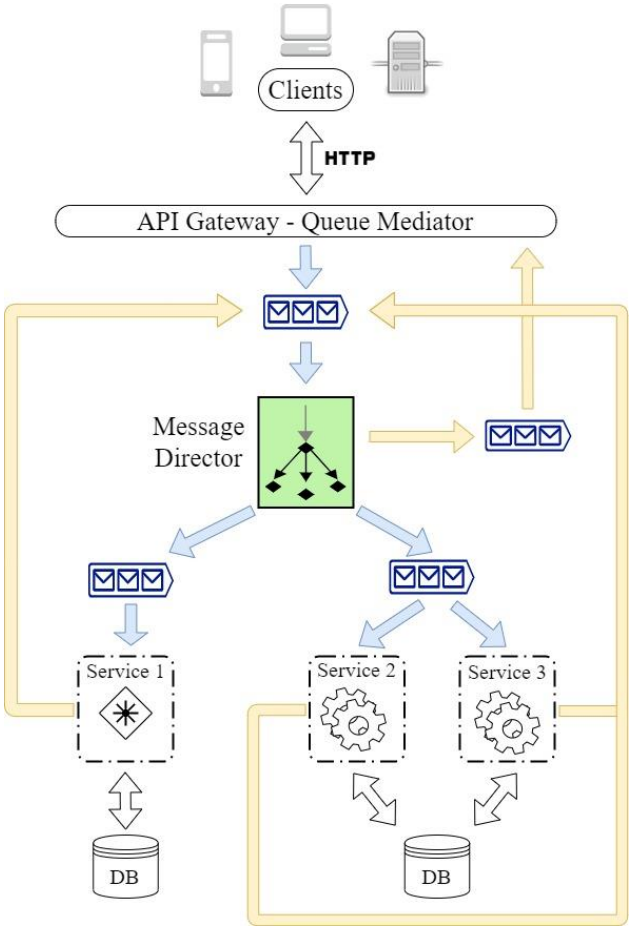



Figure 4. Proposed Enterprise Software Architecture Design

The outcomes and the real environment of the implemented architecture are discussed in the following

section. 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 possible problems.

### *3.1. Communication*

Enterprise software systems are built for making the operations of the 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.

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 [49]. 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 processed 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 the client and the server is blocked until the service returns the corresponding response of the incoming request. Conversely, the client is not blocked while waiting for the reply in asynchronous communication.

#### *3.1.1. API Gateway*

Each backend system must provide a common facade regardless of the architecture of a system to make accessibility and integration easy. There is no easy way to manage the interaction points of the enterprise systems. In compliance with the microservices approach and not to be considered as an antipattern [24] of MSA, 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.

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 the 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 [50, 51]. 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 quick development of web services, it is pervasive, and it is almost used as the default communication protocol for MSA based applications in the 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 the use of service methods by providing an interface. We aimed to keep the 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 the client. We called the method name as the intent of the client.
- Identification and authentication operations are handled inside API.

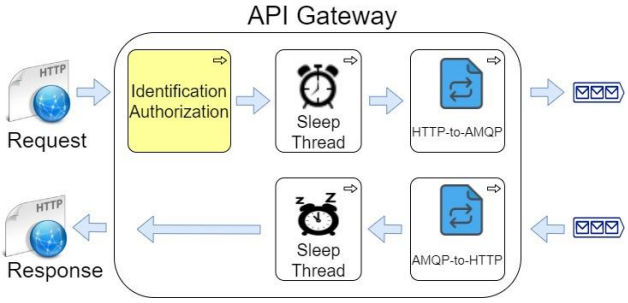



Figure 5. Request lifecycle in API Gateway

As it is shown in Figure 5, 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 the posted method name. The final task of the API after the invocation of the received event is writing the generated JSON message to a queue.

### *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, such as XML, HTML, Protocol Buffer and JSON (which is the most favorite). Protocol buffer, JSON and XML can be alternatives for the 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 an unnecessary increase in the size of transferred data. JSON is less verbose according to XML. It decreases the data size by 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-service Communication*

In MSA, all the microservices are applications that are running on their own and they must have a messaging network to communicate with internal or external applications [49, 18]. In our proposed design, we offer a synchronous RESTful-based API gateway to manage the outer interactions. There are two options to handle inter-process communication (IPC). The microservices can communicate over a synchronous request/response principle like our API gateway is 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 transactions in an enterprise system do not require sets of fully synchronized operations. The asynchronous messaging allows processing a large volume of data when the client-side does not expect an immediate response [24]. Just as a well-designed asynchronous communication can pretend to work as if it is synchronous. The vice-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 [52] as the message broker since it is the most used one and it implements the AMQP. Besides, taking responsibility for load balancing with already implemented distribution algorithms, this communication type makes the system more resilient to failure by keeping messages in queues in downtimes of the system. It also provides the state of the messages [24]. Furthermore, it eases scaling by supporting a 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 on 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 6. Private queue usage for inter-microservices communication.

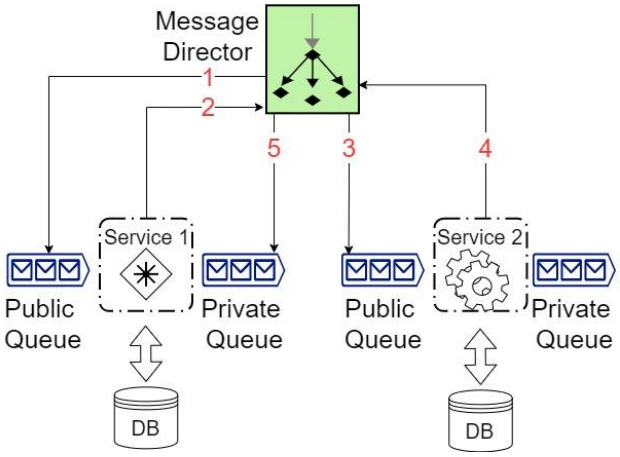



Figure 6. Private queue usage for inter-microservices communication

### *3.2. Service Registry and Discovery*

Despite all the benefits MSA offers, excessive challenging development tasks are accompanied by these benefits due to the dynamic nature of distributed systems [14, 15, 20, 45, 53, 54]. While dividing the system into smaller applications can ease the management of the 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).

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 seamless service discovery. 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 [39] is an obligation which is revealed from the effort of microservices to keep the dependencies of the services loosely coupled. Service discovery fundamentally is an ability to find all the services each other at run-time.

We implement a message router component with the name of "Message Director". The API gateway 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 the 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 the global cache manager, an exception is thrown to inform the client about the situation.

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*

The basic principle lies under MSA is "divide and conquer" [14] 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 downsides as much as possible. The study in [29] 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 [46] are two key points which should be considered to maximize the upsides. These approaches identify the way which makes the change of a microservice easier and faster. 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 [15, 55] 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 [46] 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 MSA 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 application generates a proper response to each request and publishes 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 the service registration process, all the related client request can be forwarded to the correct microservice by message director.

### *3.4. Security*

Security is a major challenge in distributed systems. The key benefits of distributed systems or MSA such as granularity, easy deployment, inter-service communication result in new security gaps and specifically, small pieces of MSA expand the security risk surface [28, 56, 57, 58]. Each IT system promotes a security layer according to the sensitivity of its content or operations.

Though the security of an organizational system has been combined as multiple security levels, protection efforts

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 the 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.

All organizational systems should identify the client of an 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 the abstraction layer of our proposed design, API gateway is the entry point of microservices-based software and it should be the first defending layer.

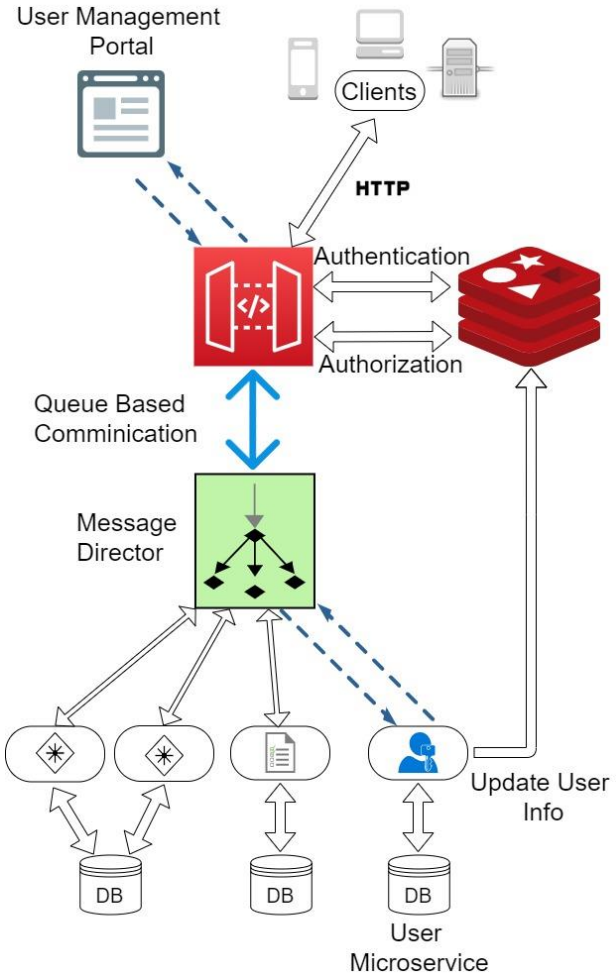



Figure 7. Authentication and authorization flow

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.

In Figure 7. 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 their access rights from server-side. Afterwards, all kind of authorization changes triggers 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.

### *3.5. Resiliency*

The health of the overall software system depends on the health of the network, DNS, data storage, virtual or physical hardware, the instances of services. When the infrastructural outages are considered out of concern in this paper, there are four best practices such as timeouts, bounded retries, circuit breakers and bulkheads [62] to achieve designing a reliable MSA.

Timeout is a determined duration in advance for every API call to guarantee that every API request is going to be replied in a specified duration. In our design, each API request thread, which is slept in API gateway while waiting for its response message, has a lifetime as the timeout value. By that way, unless the sleeping request gets the corresponding result up to the end of its lifetime, it will be awakened by the API thread pool manager and will be replied with a message declaring that the expected response message cannot be received in the acceptable duration.

Bounded retry is a pattern which retries the failed API calls complying with a determined frequency strategy. This pattern is applied to minimize the negative effects of the transient outage of the backend services. In our MSA design, this pattern is not implemented, and it is expected that the clients take over this mission. On the other hand, it is strongly recommended that the service method can be coded by considering the possibility of retried calls.

Circuit breaker is a method which is designed to handle the repeated failed calls of a microservice. Circuit breaker tracks all the requests and their responses to detect a problem regarding the health of a microservice. When a problem is detected, the circuit breaker turns to open-mode and tries to transfer the request calls of the problematic microservice to any alternative microservice if there is described one or generates a default response.

When the circuit breaker is in the open mode, it periodically checks the status of the down microservice. If its connection attempts are successful, then it turns to close mode to settle the connection as its normal running mode [45, 62].

For the sake of increasing resiliency, Circuit-Breaker pattern is implemented as a component of Message Director to forward messages automatically to a predefined alternative handler when an integration point is down until the basic service provider gets back into circulation.

Bulkhead pattern is designed to tolerate failure and is inspired from bulkheads of a ship's hull [63]. The principle aims to provide a partitioning over the isolated access channels of the microservices. When a microservices is down, the pattern ensures not to allow any blockage in communication or a resource scarcity for consuming services.

The purpose of this pattern is served partially with the timeout operation that is taken places in the API gateway. Besides, the time-to-live argument of RabbitMQ messages is set to guarantee that the expired messages will be removed from the queue of the down microservices. So, RabbitMQ resources are not consumed for dead messages.

### *3.6. Database Selection*

Data persistency is one of the most important and expensive tasks for any application. MSA allows each microservice instance having an individual data management solution such as fully supported transactional databases, open-source and supported on-demand databases, document-based databases or the combinations of those in compliant with the requirements of the task which microservice operates. One can decide how to handle data persistency with regards to the planned budget, development team experience, whether business requirements must be operated transactional, the amount of the data to be persisted and what the speed expectation of querying time [64]. 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, the 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.

### *3.7. Architectural Comparison*

After designing the proposed enterprise microservices-based software architecture, the features of the proposed architecture are compared with the existing microservices frameworks [27, 64] in Table 1.

The overall picture depicted from Table 1 shows that our proposed design handles the cross-cutting concerns of MSA. However, there are functionalities which are covered by other frameworks need to be implemented in our design such as service governance, remote configuration and richer serialization support. In addition to these, an advance level load balancer is required to balance the heavy load over multiple instances of API gateway.

Table 1. Microservices architecture comparison

| Feature | Microservices Architecture | | | |
|---|---|---|---|---|
| | *Dubbo* | *Vert.x* | *Spring Cloud* | *Our Design* |
| Service Interface | RPC /RESTful | RESTful | RESTful | RESTful/ AMQP |
| Automatic Service Registry-Discovery | ✓ | ✓ | ✓ | ✓ |
| Security | OAuth2 | OAuth2 | OAuth2 | OAuth2 |
| Load Balancer | ✓ | ✓ (when service-bus used) | ✓ | ✓ (RabbitMQ provides only for microservices) |
| Circuit Breaker | ✓ | ✓ | ✓ | ✓ |
| Service Governance | ✓ | ✓ | ✓ | - |
| Remote Configuration | ✓ | ✓ | ✓ | - |
| Distributed Logging | ✓ | - | - | ✓ |
| Large Data Handling | ✓ | - | ✓ | - |
| Containerization | ✓ | ✓ | ✓ | ✓ |
| Serialization | ✓ | ✓ | ✓ | JSON, AMQP |
| Reactive Programming | - | ✓ | - | ✓ |

## 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 **Error! Reference source not found.**2. 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 were deduced from JMeter performance calculations. RabbitMQ server is used as a message broker and Redis is used as a global cache

manager. IIS 8 is the webserver at which the RestAPI is deployed and served.

Table 2. 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 an 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 that MSA has a higher network delay in comparison to the monolith architecture. As the additional latency of message director and message broker is regarded, the reason for the delay can be inferred. We measure that average round-trip time is 25 ms for monolith and 30 ms for our MSA prototype. The latency delta is about 5 ms per request.

Table 3. 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 |

Each MSA instance creates a new thread for handling each received request. Therefore, 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 **Error! Reference source not found.**, 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.

We test the performance of the AMQP comparing with the performance of the HTTP RestAPI. The results in **Error! Reference source not found.** 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 with a response for one 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.

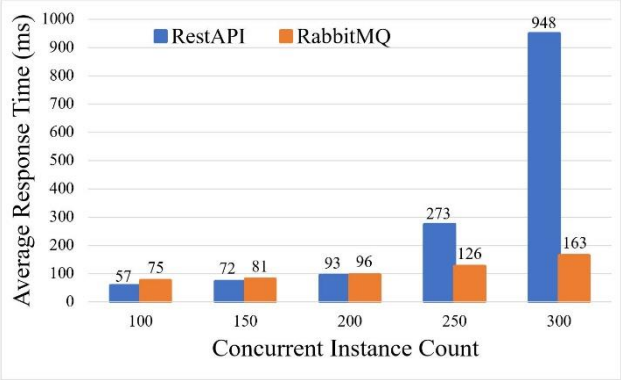



Figure 8. RabbitMQ and HTTP RestAPI performance comparison

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 **Error! Reference source not found.**, it is clearly seen that, when the number of concurrently running instances increase, the MSA allows a reduction in the response time proportionally similar to the monolith architecture.

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 a 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.

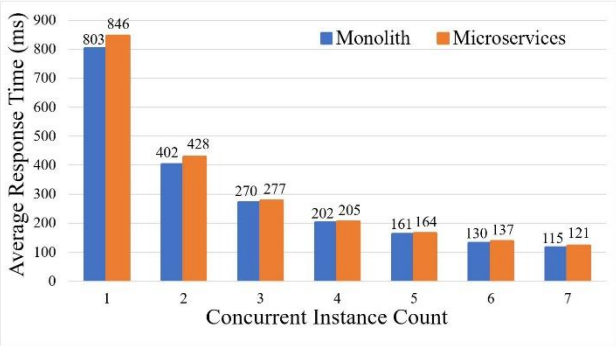



Figure 9. Bubble Sort Response time for an integer array of 10000 items while instance count increase

We also measure the number of messages handled during the execution of bubble sort operation. **Error! Reference source not found.** shows a gradual increase in the handled number of messages per second.

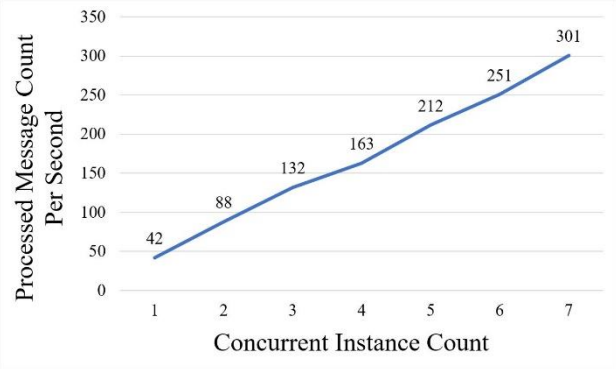



Figure 10. Message processing velocity for Figure 9 test case

## 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. The shared design details cover more microservices concerns, preserve clearer architectural perspective according to previous related work. We implement a prototype of designed architecture based on microservices and give architectural data flow and control mechanism details. As mentioned in some references which are related to migration from monolith to MSA, the easiest way to gauge the success of the migration project is to compare the performance of the MSA with the performance of former monolithic architecture to see whether the microservices can meet or transcend. Thus, we 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 the 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 the 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 the current state of the system, measuring transaction times, capturing business or infrastructure errors to report.

For future work, a rule-based or even a learning tracking and monitoring tool can be designed using an 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. A trained or a well-designed message tracking and monitoring tool may allow taking autonomous proactive actions in case an inconsistent state of the MSA.

Service governance mechanism should be implemented by preserving metrics and diagnostic data from each component of the proposed MSA. The serialization methods of the system can be enriched to support more data formats. To leverage load balancing operation, a load balancer can be developed at the API gateway level. The designed architecture can only use AMQP for inter-service communication. HTTP support should be done, and a lightweight HTTP load balancer can be developed in the message director component.

## REFERENCES

[1] Internet: M. Fowler, J. Highsmith, The Agile Manifesto, http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf, 26.02.2020.

[2] Internet: E. Mueller, The Agile Admin, https://theagileadmin.com/what-is-devops/, 24.01.2020.

[3] Internet: M. Rose, Teach Target, https://searchcloudcomputing.techtarget.com/definition/cloud-computing, 14.02.2020.

[4] R. V. O'Connor, P. Elger, P. M. Clarke, "Continuous software engineering—A microservices architecture perspective", *Software: Evolution and Process*, 29(11), 1-12, 2017.

[5] J. Bosch, **Continuous Software Engineering**, Springer, Switzerland, 2014.

[6] D. Saff, M. D. Ernst, "An Experimental Evaluation of Continuous Testing During Development", *ACM SIGSOFT Software Engineering Notes*, 29(4), 76-85, 2004.

[7] D. Saff, M. D. Ernst, "Reducing wasted development time via continuous testing", **14th International Symposium on Software Reliability Engineering**, Denver, USA, 281-292, 17-20 November, 2003.

[8] M. Virmani, "Understanding DevOps & Bridging the Gap from Continuous Integration to Continuous Delivery", **Fifth International Conference on the Innovative Computing Technology**, Pontevedra, Spain, 78-82, 20-22 September, 2015.

[9] C. M. Aderaldo, N. C. Mendonça, C. Pahl, P. Jamshidi, "Benchmark Requirements for Microservices Architecture Research", **2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE***)*, Buenos Aires, 8-13, 20-22 May, 2017.

[10] T. Ueda, T. Nakaike, M. Ohara, "Workload Characterization for Microservices", **2016 IEEE International Symposium on Workload Characterization (IISWC),** Providence USA, 1-10, 25-27 September, 2016.

[11] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, M. Steinder, "Performance Evaluation of Microservices Architectures Using Containers", **2015 IEEE 14th International Symposium on Network Computing and Applications**, Cambridge, USA, 27-34, 28-30 September, 2015.

[12] F. Wang, F. Fahmi, "Constructing a Service Software with Microservices", **2018 IEEE World Congress on Services (SERVICES),** San Francisco**,** USA, 43-44, 2-7 July, 2018.

[13] H. Knoche, W. Hasselbring, "Experience with Microservices for Legacy Software Modernization", *Software Engineering and Software Management*, 292, 101-102, 2019.

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

[15] Internet: J. Bonér, D. Farley, R. Kuhn, M. Thompson, The Reactive Manifesto, https://www.reactivemanifesto.org/, 20.02.2020.

[16] J. Bonér, **Reactive Microsystems The Evolution of Microservices at Scale**, Lightbend, USA, 2017.

[17] J. Bogner, A. Zimmermann, "Towards Integrating Microservices with Adaptable Enterprise Architecture", **2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW),** Vienna, Austria, 1-6, 5-9 September, 2016.

[18] M. Söylemez, A. Tarhan, "Mikroservis Mimarisi ve Mimari Faktörleri Üzerine Endüstriyel Bir İnceleme", **Proceedings of the 12th Turkish National Software Engineering Symposium,** Istanbul, Turkey, 1-13, 10-12 September, 2018.

[19] Y. Yu, H. Silveira, M. Sundaram, "A microservice based reference architecture model in the context of enterprise architecture", **2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference** , China, 1856-1860, 3-5 October, 2016.

[20] A. Yamuç, U. M. Sürme, "Uydu Yer Yazılım Sistemleri için Servis-yönelimli Mimari'den Mikroservis Mimarisine Geçiş Stratejisi", **Proceedings of the 7th Turkish National Software Architecture Conference (UYMK 2018)**, Istanbul, Turkey, 1-12, 29-30 November, 2018.

[21] W. Tang, L. Wang, G. Xue, "Design of Information System Architecture of Garment Enterprises Based on Microservices", *Journal of Physics: Conference Series*, 1168(3), 32128-32135, 2019.

[22] C. Pinheiro, A. Vasconcelos, S. Guerreiro, "Microservice Architecture from Enterprise Architecture Management Perspective", *Lecture Notes in Business Information Processing (LNBIP)*, 356, 236-245, 2019.

[23] L. Huang, C. Zhang, Z. Zeng, "Design of a public services platform for university management based on microservice architecture", *Microsyst Technologies*, 1-6, 2019.

[24] A. Akbulut, H. G. Perros, "Performance Analysis of Microservice Design Patterns", *IEEE Internet Computing*, 23(6), 19-27, 2019.

[25] Internet: Spring, Spring Cloud, https://spring.io/projects/spring-cloud, 27.04.2020.

[26] Internet: Vert.x, Eclipse Vert.x is a tool-kit for building reactive applications on the JVM, https://vertx.io/, 29.04.2020.

[27] Internet: Dubbo, A high performance Java RPC framework, https://dubbo.apache.org/en-us/, 30.04.2020.

[28] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, "Microservices: Yesterday, Today, and Tomorrow", **Present and Ulterior Software Engineering**, M. Mazzara, B. Meyer, Springer International Publishing, Zurich, Switzerland, 195-216, 2017.

[29] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures", **16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)**, Cartagena, Colombia, 179-182, 16-19 May, 2016.

[30] A. Messina, R. Rizzo, P. Storniolo, A. Urso, "A Simplified Database Pattern for the Microservice Architecture", **The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications**, Lisbon, Portugal, 35-40, 26-30 June, 2016.

[31] Internet: A. Nadalin, On monoliths, service-oriented architectures and microservices, https://odino.org/on-monoliths-service-oriented-architectures-and-microservices/, 13.03.2020.

[32] Internet: D. Anastasia, Monolith, SOA, Microservices, or Serverless?, https://rubygarage.org/blog/monolith-soa-microservices-serverless, 01.03.2020.

[33] Internet: S. Arshed, Monolithic vs SOA vs Microservices—How to Choose Your Application Architecture, https://medium.com/@saad_66516/monolithic-vs-soa-vs-microservices-how-to-choose-your-application-architecture-1a33108d1469,18.03.2020.

[34] Internet: The Open Group, Service-Oriented Architecture What Is SOA, http://www.opengroup.org/soa/source-book/soa/p1.htm#soa_definition, 23.04.2020.

[35] M. Richards, **Microservices vs. Service-Oriented Architecture**, O'Reilly Media, CA, USD, 2016.

[36] Internet: E. Evans, GOTO 2015 - DDD & Microservices: At Last, Some Boundaries!,https://www.youtube.com/watch?v=yPvef9R3k-M, 23.04.2020.

[37] Internet: J. Lewis, M. Fowler, Microservices, https://martinfowler.com/articles/microservices.html, 26.04.2020.

[38] Internet: R. C. Martin, The Clean Code Blog, https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html, 24.04.2020.

[39] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, A. D. Salle, "Towards Recovering the Software Architecture of Microservice-Based Systems", **2017 IEEE International Conference on Software Architecture Workshops**, Gothenburg, Sweden, 46-53, 5-7 April, 2017.

[40] W. Hasselbring, G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce", **2017 IEEE International Conference on Software Architecture Workshops (ICSAW)**, Gothenburg, Sweden, 243-246, 5-7 April, 2017.

[41] D. Shadija, M. Rezai, R. Hill, "Towards an Understanding of Microservices", **2017 23rd International Conference on Automation and Computing (ICAC),** Huddersfield, UK, 1-6, 7-8 September, 2017.

[42] O. Zimmermann, "Microservices tenets", *Computer Science - Research and Development*, 32(3), 301-310, 2017.

[43] D. Taibi, V. Lenarduzzi, C. Pahl, "Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation", *IEEE Cloud Computing*, 4(5), 22-32, 2017.

[44] H. Kang, M. Le, S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps", **2016 IEEE International Conference on Cloud Engineering (IC2E)**, Berlin, Germany, 202-211, 4-8 April, 2016.

[45] B. Butzin, F. Golatowski, D. Timmermann, "Microservices approach for the internet of things", **2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)**, Berlin, Germany, 1-6, 6-9 Semtember, 2016.

[46] Internet: F. Montesi, J. Weber, Circuit Breakers Discovery and API Gateways in Microservices, http://arxiv.org/abs/1609.05830, 30.04.2020.

[47] S. Newman, Building Microservices Designing Fine-Grained **Systems**, O'Reilly Media, Sebastopol, CA, 2015.

[48] D. Namiot, M. Sneps-Sneppe, "On Micro-services Architecture", *International Journal of Open Information Technologies*, 2(9), 24-27, 2014.

[49] E. Wolff, **Microservices-Flexible Software Architecture, Crawfordsville**, Pearson Education, Indiana, USA, 2017.

[50] Internet: C. Richardson, Building Microservices: Inter-Process Communication in a Microservices Architecture, https://www.nginx.com/blog/building-microservices-inter-process-communication, 30.04.2020.

[51] Internet: D. Jacobson, Why REST Keeps Me Up At Night, https://www.programmableweb.com/news/why-rest-keeps-me-night/2012/05/15, 30.04.2020.

[52] Internet: M. Rouse, REST (REpresentational State Transfer), https://searchmicroservices.techtarget.com/definition/REST-representational-state-transfer, 30.04.2020.

[53] Internet: RabbitMQ, RabbitMQ is the most widely deployed open source message broker, https://www.rabbitmq.com/, 30.04.2020.

[54] Internet: C. Richardson, Service Discovery in a Microservices Architecture, https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture, 30.04.2020.

[55] S. Sobernig, U. Zdun, "Inversion-of-Control Layer", **Proceedings of the 15th European Conference on Pattern Languages of Programs (EuroPLoP'10)**, Irsee, Germany, 1-22, 7-11 July, 2010.

[56] Internet: R. Chandramouli, Security Strategies for Microservices-based Application Systems, https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204-draft.pdf, 30.04.2020.

[57] Y. Sun, S. Nanda, T. Jaeger, "Security-as-a-Service for Microservices-Based Cloud Applications", **2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)**, Vancouver, Canada, 50-57, 30 November-3 December, 2015.

[58] Internet: D. Yu, Y. Jin, Y. Zhang, X. Zheng, A survey on security issues in services communication of Microservices-enabled fog applications,https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.4436, 30.04.2020.

[59] Internet: Sumo Logic, Improving Security in Your Microservices Architecture, https://www.sumologic.com/insight/microservices-architecture-security/, 30.04.2020.

[60] Internet: OAuth, OAuth 2.0, https://oauth.net/2/, 30.04.2020.

[61] Internet: Auth0, Introduction to JSON Web Tokens, https://jwt.io/introduction, 30.04.2020.

[62] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices", **2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)**, Nara, Japan, 57-66, 27-30 June, 2016.

[63] Internet: Microsoft, Bulkhead Pattern, https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead, 28.04.2020.

[64] E. Edling, E. Östergen, **An analysis of microservice frameworks**, Bachelor, Linköping University, Department of Computer and Information Science, 2017.