T.C.
BEYKENT ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

FARKLI SERVİS ODAKLI YAKLAŞIMLARIN PERFORMANS DEĞERLENDİRMELERİ

Yüksek Lisans Tezi

Tezi Hazırlayan:

GÖKHAN MERDEN

İstanbul, 2022

T.C.
BEYKENT ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

FARKLI SERVİS ODAKLI YAKLAŞIMLARIN PERFORMANS DEĞERLENDİRMELERİ

Yüksek Lisans Tezi

Tezi Hazırlayan:

GÖKHAN MERDEN

1908020003

Orcid: 0000-0002-3736-8955

DANIŞMAN

Dr. Öğr. Üyesi Ediz ŞAYKOL

İstanbul, 2022

YEMİN METNİ

Yüksek Lisans Tezi olarak sunduğum “Farklı Mimari Yaklaşımların Performans Kıyaslamaları” başlıklı bu çalışmanın, bilimsel ahlak ve geleneklere uygun şekilde tarafımdan yazıldığını, yararlandığım eserlerin tamamının kaynaklarda gösterildiğini ve çalışmamın içinde kullanıldıkları her yerde bunlara atıf yapıldığını belirtir ve bunu onurumla doğrularım. 29.04 2022

Gökhan MERDEN

T.C.
BEYKENT ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ MÜDÜRLÜĞÜ
TEZLİ YÜKSEK LİSANS SINAV TUTANAĞI

29/04/2022

Enstitümüz ***Bilgisayar Mühendisliği*** Anabilim Dalı ***Bilgisayar Mühendisliği*** Programı Yüksek Lisans öğrencilerinden ***1908020003*** numaralı ***Gökhan MERDEN***'in "Beykent Üniversitesi Eğitim-Öğretim Yönetmeliği"nin ilgili maddesine göre hazırlayarak Enstitümüze teslim ettiği ***"Farklı Servis Odaklı Yaklaşımların Performans Değerlendirmeleri"*** konulu tezini, Yönetim Kurulumuzun 11/01/2022 tarih ve 2022/02 sayılı toplantısında seçilen ve On-Line toplanan biz jüri üyeleri huzurunda, Beykent Üniversitesi Lisansüstü Eğitim ve Öğretim Yönetmeliğinin 29. maddesinin 3. fıkrası gereğince 45 dakika süre ile Zoom programı aracılığıyla on-line olarak aday tarafından savunulmuş ve sonuçta adayın tezi hakkında ***"OYBİRLİĞİ"*** ile ***"KABUL"*** kararı verilmiştir.

İşbu tutanak, 2 nüsha olarak hazırlanmış ve Enstitü Müdürlüğü'ne sunulmak üzere tarafımızdan düzenlenmiştir.

DANIŞMAN
Dr. Öğr. Üyesi Ed*** ŞA***
(Beykent Üniversitesi)

ÜYE
Dr. Öğr. Üyesi At*** YI***
(Beykent Üniversitesi)

ÜYE
Dr. Öğr. Üyesi Öm*** ÇE***
(Milli Savunma Üniversitesi)

Adı Soyadı : Gökhan MERDEN
Danışmanı : Dr. Öğr. Üyesi Ediz ŞAYKOL
Türü ve Tarihi : Yüksek Lisans Tezi, 2022
Alanı : Bilgisayar Mühendisliği
Anahtar Kelimeler : Mikroservis, Mikroservis mimarisi, SOA, Docker, Performans, internet trafiği

ÖZ

FARKLI SERVİS ODAKLI YAKLAŞIMLARIN PERFORMANS DEĞERLENDİRMELERİ

Gelişen teknoloji ile değişen ihtiyaçlar, yazılım geliştirme projelerini ve onların kullandıkları teknolojileri de etkilemiştir. İnternet kullanımının artmasıyla insanların ihtiyaçlarının büyük bölümünü internet ortamından çözüme kavuşturmaya başlamışlardır. Sağlık alanındaki gelişmeler, e-ticaret sektöründeki gelişmeler yazılım sektörü ile iç içe geçmiştir.

Mobil uygulamaların kullanımının artmasıyla ve Iot (internet of things) kullanımının yaygınlaşmasıyla yazılım alanında dağıtık mimari yaklaşımının da değişmesine yol açmıştır. İnternet ortamında dolaşan veri trafiğinin artması ile daha düşük veri ile daha yüksek performans sağlayacak araç ve gereçleri kullanma konusunda düşünmeye sevk etmiştir.

90 'lı yıllarda kullanıma başlanan dağıtık mimari olan SOA, ihtiyaç duyulan veri miktarının artması ve artan cihaz çeşitliliği nedeniyle performans sorunlarını beraberinde getirmiştir. Mobil cihazların ve e-ticaret sistemlerinin kullanımındaki zorluklar, geliştiricileri yeni teknolojileri kullanmaya sevk etmiştir. Ayrıca veri yönetim sistemlerinin merkezi yapıdan çıkıp çok merkezli hale gelmesi kurumsal ölçekli büyük uygulamaların kullanımını ve geliştirilmesini zorlaştırmıştır.

2000' li yılların ortasına doğru ortaya çıkan Mikroservis mimarisi, değişen ihtiyaçlar ve artan cihaz çeşitliliği ile daha kolay bağlantı yapma ve daha düşük veri transferi sağlaması neticesinde kullanımı yaygınlaşmıştır. Mobil cihazların

kullanımının artması ile bu cihazların Mikroservisler ile kolay bağlantı kurabiliyor olması teknoloji şirketlerinin de geliştirmelerini bu teknolojiyi kullanarak ilerlemeye sevk etmiştir.

Bununla birlikte Google tarafından geliştirilen açık kaynaklı gRPC altyapısının artan veri miktarı ile ortaya çıkan performans sorunlarına çözüm için yeni bir bakış açısı getirmiştir. Ağ trafiğindeki transfer edilen veri miktarı düşürülmeye çalışılmış, böylece performans artışı sağlanması hedeflenmiştir.

Yapılan bu çalışmanın amacı SOA, Mikroservisler ve gRPC teknolojilerinin kıyaslamasını yaparak nasıl performans kazanımları elde edilebileceğini anlamaya yardımcı olmaktır. Çeşitli alanlarda teknoloji kullanımı artarken, ihtiyaçlar değişmeye başlamıştır. Değişen ihtiyaçlara göre yeni teknolojiye geçiş yapmak oldukça maliyetli ve uzun soluklu bir süreçtir. Bu çalışma yeni teknolojiye geçiş yapmanın getirdiği performans kazanımlarını incelemektedir.

Name Surname : Gökhan MERDEN
Supervisor : Dr. Lecturer Ediz ŞAYKOL
Degree ve Date : Master, 2022
Major : Computer Engineering
Key Words : Microservices, Microservice architecture, SOA, Docker, Performance, internet traffic

ABSTRACT

PERFORMANCE EVALUATIONS OF DIFFERENT SERVICE ORIENTED APPROACHES

Software development projects are effected by changing needs while technology is improving. People have started to solve most of their needs by internet. Improvements on health or e-commerce sectors are intertwined with software development by changing technology.

Distributed architecture approach is evolving with usage of mobil apps or Iot. Data traffic is increasing on internet so tools which are less data and high performance are in mind of development teams. Because existing enterprise-scale systems could not be compatible with new technology devices.

SOA which is distributed architecture is started to use in 90's. While device variety and data traffic is increasing , it brought performance problems together. Mobile devices and e-trade systems encountered some difficulties. So development teams are motivated to use new technologies. Furthermore the fact that data management systems leave centered structure and become multi-centered has made it difficult to use and develop enterprise-scale applications.

Microservice architecture which emerged towards the middle of 2000s has become widespread as a result of changing needs and device diversity. Also lower data transfer and establishing easier connections are another effects to use by developer teams. Microservices can transfer data with mobile devices which is

increasing usage day by day easily. So technology companies motivated to develop with new technologies

However open source gRPC framework which is developed by Google has brought a new method to solve performance problems while increasing amount of data. Amount of data in network traffic has been tried to be reduced. Thus it is aimed to increase performance.

The purpose of this study is to help understand how performance gains can be achieved by comparing SOA, Microservices and gRPC Technologies. While the use of technology in various fields has increased, needs have begun to change. it examines the performance gains brought by the transition to new technology.

İÇİNDEKİLER

Sayfa No.

TABLOLAR LİSTESİ

Sayfa No.

ŞEKİLLER LİSTESİ

Sayfa No.

KISALTMALAR

BFF : Backend For Frontend

CI/CD : Continious Integration / Continious Development

HTTP : Hyper-text Transfer Protocol

IOT : Internet Of Things

JSON : JavaScript Object Notation

PAAS : Platfrom As A Service

REST : Representational State Transfer

SOA : Service Oriented Architecture

UFS : Union File System

WSDL : Web Services Description Language

XML : Extensible Markup Language

SÖZLÜK

Ağ Trafiği: Belirli bir zamanda bir ağda hareket eden veri miktarını ifade eder.

Yük Testi: Uygulamaya giderek artan sayıda kullanıcı ile yüklenerek maksimum düzeyi belirler.

Sanallaştırma Teknolojisi: fiziksel bir bilgisayardaki bellek, işlemci ve hafıza gibi kaynakların birbirinden farklı birden fazla sanal makineler (virtual machines) ile eş zamanlı kullanım imkânı sağlayan bir teknolojidir.

Test Ortamı: Uygulama yayınlanmadan önce olası sorunların tespiti için oluşturulan ortamdır.

Çıktı Metrikleri: Geliştirilen uygulamadan elde edilen sonuçlar olarak ifade edilir.

Yük Dengeleyici: 2 veya daha fazla servis arasında iş paylaştırma işlemi yapan uygulamadır.

Serileştirme İşlemi: Bir veri yapısını iletilebilecek veya saklanabilecek formata çevirme işlemidir.

GİRİŞ

Teknolojinin gelişmesiyle birlikte dağıtık mimari ile geliştirilen kurumsal uygulamalar zamanla çağın ihtiyaçlarına uygun gelişim gösterme arayışları içinde olmuşlardır. Gelişen teknolojiye uyum sürecinde ortaya çıkan zaman ve maddi maliyetler yazılım ekiplerini değip değmeyeceği konusunda düşünmeye sevk etmiştir. Diğer bir taraftanda teknolojinin geliştiği farklı alanlarda yazılım dünyasını gelişmeye sevk etmiştir. Günümüzde mobil telefonların yaygınlaşmasından, sosyal medya uygulamalarının daha yoğun kullanmasından, sağlık alanındaki gelişmelerle entegre olacak şekilde ortaya çıkan ihtiyaca cevap vermek için daha hızlı, daha verimli ve daha az ağ trafiği yaratan çözümlere yönlenilmiştir.

Mobil telefonların kullanmının yaygınlaşması, Iot' nin hayatın bir parçası olmaya başladığı bu günlerde bu cihazlarla kolay, etkin ve hızlı şekilde veri alışverişi yapabilecek teknolojilere ihtiyaç duyulmuştur. Sağlık verilerinin anlık olarak kayıt edilip takip edilmesi, ekonomik verilerin anında ilgili yerlere ulaştırılması önem arzettiğinden, kaçınılmaz olarak değişen ihtiyaca uygun olarak yeni teknolojilerin kullanılması ihtiyaç haline gelmiştir. Geçmişten günümüze gelişen teknoloji ile çeşitli servis tabanlı mimari yaklaşımları düşünülerek uygulamaya geçilmiştir.

Bu teknolojileri inceleyecek olursak;

Servis tabanlı mimari, çok daha eski yıllara dayanmakla birlikte 90 lı yıllarda ortaya çıkmış ve kullanılmaya başlanmış bir teknolojidir. Uçtan uca erişim sağlayarak birbirine gevşek bağlı kapsamlı kurumsal ve ölçeklenebilir sistemler geliştirilmesine olanak sağlamıştır. Büyük ölçekli problemleri parçalara ayırarak ve bu parçalara erişilebilir adresler verilerek bu problemlerin çözülmesi için bir başlangıç noktası haline gelmiştir.

Servis tabanlı mimari, sistemi küçük parçalar ile sınırlayan bir model sunmaktadır. Daha büyük iş parçaları küçük parçalar içerebilir. Bu parçalar dağıtık olabilir.

SOA birbirinden bağımsız iş parçalarının bir standart ve kurallar çerçevesinde var olmasına yardımcı olur. Bu küçük parçalar SOA içinde bir servis olarak bilinir.

2000'li yıllara gelindiğinde gelişen ihtiyaçlar çerçevesinde Representational State Transfer (REST) teknolojisi geliştirilmiştir. Bu teknolojinin varolan özelliklerinin bir bölümü SOA'dan halef almakla beraber daha basit, hızlı ve durumsuz bağlantı sağlamaktadır. Böylelikle farklı işletim sistemi kullanan, daha az donanım gücüne sahip olan, daha az ağ trafiği oluşturan cihazlar ile etkileşim daha kolay hale gelmiştir.

Mikroservis mimarisinin veri gönderimi (POST), veri çekme (GET), veri güncelleme (PUT), veri silme (DELETE) gibi metodları bulunmaktadır. Bu metodlar SOA'nın büyük xml dökümanı yerine daha küçük boyutlu JSON döküman ile yapabilmektedir.

Bu özelliği ile SOA'ya göre çok daha performanslı işlem yapılabilmektedir. Günümüzde her ne kadar internet bağlantı hızları oldukça yüksek hızlara ulaşabilsede, yapılan işlem miktarı çok daha fazla artmıştır. Buna gelişen akıllı telefon teknolojisi ve mobil uygulamaların kullanımının yaygınlaşmasının oldukça fazla etkisi olmuştur.

Bununla birlikte Mikroservis mimarisinin kullanımının yaygınlaşması ile artan ihtiyaç ile orantılı bir biçimde sunucu maliyetlerine de yansımıştır. Akabinde uygulamaları ve uygulama bağımlılıklarını da içeren bir paket haline getirip çalıştırılabilir bir ortam sunan Docker ile sunucular daha verimli ve etkin kullanılmaya başlanmıştır.

Mikroservis mimarisinin yoğun kullanımı ve artan veri trafiği beraberinde farklı sorunları da getirmiştir. Mikroservis, SOA'nın kullandığı XML veri tipine karşılık JSON veri tipini kullanmaktadır. Böylelikle daha az veri daha hızlı iletilmektedir. Ne var ki teknolojinin her alanda kullanılması ile artan veri yoğunluğunu taşımak için farklı teknoloji firmalarını farklı çözümler aramaya sevk etmiştir.

Mikroservisler, JSON tipinde veri tutarak, XML veri tipinden daha küçük boyutlara sahip olduğundan bahsedilmiştir. Gün geçtikçe artan veri miktarı neticesinde JSON veri tipi ile elde edilen veri boyutu oldukça büyük boyutlara

gelmeye başlamıştır. Tabii burada artan veri boyutunun transferinde süreler artmaya başlamaktadır.

Google firmasının açık kaynak kodlu olarak geliştirdiği gRPC altyapısı devreye girerek SOA' nın kesin yapısı ile Mikroservislerin düşük boyutlu yapısını birleştirerek oldukça makul boyut ve sürelerde veri transferine olanak sağladığı görülmektedir. Mikroservisler' in veya SOA' nın istek-cevap yaklaşımına yeni bir soluk getirerek aynı bağlantı üzerinden sunucu veya istemci taraflı akış bağlantısı yaparak yeni bir yöntem ortaya koymaktadır.

Giderek artan veri boyutları ve ağ trafiğinin gelişen mobil teknolojiler ile daha fazla artması ile performans ve istek-cevap süresi çok hızlı sistemler ihtiyaç haline gelmiştir. gRPC alt yapısı Protocol Buffers tipinde ikili bayt olarak dönüştürülür ve istemciler arasında bu tiple haberleşilir. Bu teknolojinin kullanılacağı yerleri doğru belirlemek gerekir. Servisler arası veri iletişimi yapılması planlandığı durumlarda kullanımı uygundur.Ancak farklı platformlar için tam uyumlu olduğu söylenemez. Bu durum BFF uygulamalar bir ara katman olarak devreye girer ve gRPC servisine doğrudan erişemeyen istemciler bu şekilde işlemlerini tamamlarlar.

Bu tez kapsamında yapılan özgün yan bu şekilde bir çalışmanın yapılmamış oluşu ve bu şekilde yapılan çalışmalardan sonra SOA mimarisiyle geliştirme yapmış kurumların ve geliştirici takımlarının Mikroservis, Docker ve gRPC teknolojilerine geçiş yaptıklarında nasıl kazanımlar elde edecekleri ele alınmıştır.

SOA mimarisi ile Mikroservis mimarisinin ağ trafiğinde gösterdiği performans kıyaslaması yapılmış olup teknolojilerin veri aktarımı yaparken kullandığı yöntemler ve bu yöntemlerin getirdiği kazanımlardan bahsedilmiştir. Mikroservis mimarisinin Docker uygulaması ile birlikte kullanımının getirdiği avatanjlardan bahsedilmiş ve gRPC teknolojisinin Mikroservisler ile birlikte kullanılarak sağladığı düşük ağ trafiği ile veri aktarımı ele alınmıştır. Bu teknolojilerin sağladığı faydaların yanı sıra beraberinde getirdiği problemlere de değinilmiş, avatanj ve dezavatanjları ele alınmıştır.

Bu yöntemi özgün kılan nokta ise bu şekilde bir karşılaştırmanın yapılmamış olması ve bu tür bir karşılaştırma yapıldığında veri alışverişinde ne gibi kazanımlar elde edildiği, gelişen teknoloji ile ortaya çıkan araçların kullanımının nasıl etkilerinin olduğu ortaya konmuştur. Böyle bir kıyaslama yapınca düşük ağ trafiği ile daha yüksek miktarda veri alışverişi yapılabileceği böylelikle daha performanslı sistemler geliştirilebileceği gibi bilgiler elde edinilebiliyor.

Bu çalışma organizasyonu şu şekildedir:

- Bölüm 2 ‘de bu çalışmada incelenen literatür araştırmaları,
- Bölüm 3 ‘de incelenen mimariler,
- Bölüm 4 ’de performans karşılaştırması

sunulmuştur.

1. LİTERATÜRDE YER ALAN BENZER ÇALIŞMALAR

Gos ve Zabierowski (2020) çalışmalarında bir senaryo oluşturarak e-ticaret sisteminin monolitik ve mikroservis mimarilerinde geliştirme yapmışlardır. Bu çalışmalarında her iki sisteme de istek gönderilmiş ve bu isteklerin cevap süreleri karşılaştırılmıştır.

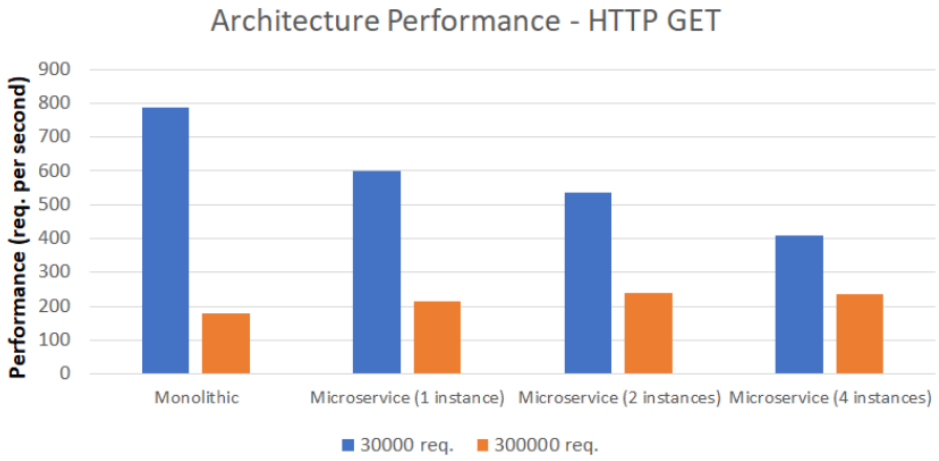
İstekleri 2 vakaya ayırarak yapmışlardır

1. 30 kullanıcı 1000 istek göndermiştir

   (30 * 1000 = 30000 istek)

2. 30 kullanıcı 1000 istek göndermiştir.

   (30 * 10000 = 300000

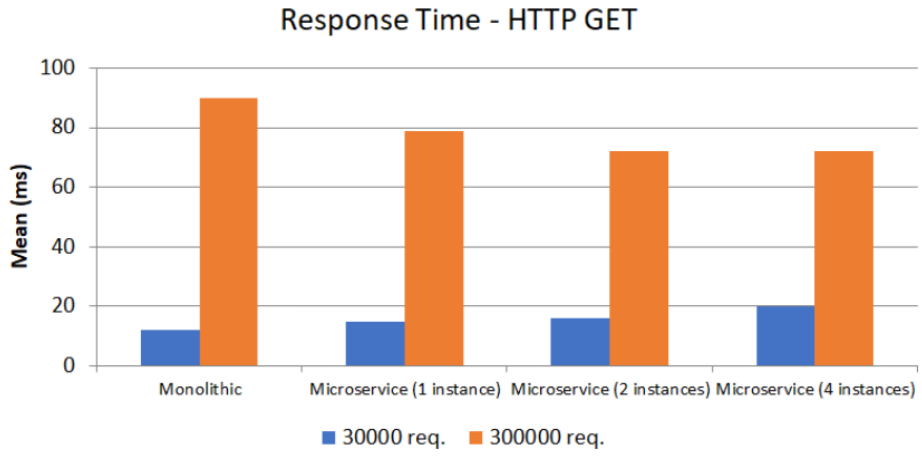


Şekil 1 HTTP Get Mimari Performans Grafiği

Kaynak: Gos ve Zabierowski (2020) 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)

Uygulamalara ilk önce tek sefer 30000 istek gönderilerek sonuçlar şekilde mavi indikatör ile gösterilmiştir. Şekilde monolitik uygulama saniyede 800/s istek alarak mikroservis uygulamasından daha fazla performans gösterdiği görülmektedir. Mikroservis uygulaması replika edilmesine rağmen en fazla 600/s civarı istek almıştır.

2. vakada ise 300000 istek gönderim işlemi yapılmıştır. Sonuçlar şekilde turuncu indikatör ile gösterilmiştir. Mikroservis uygulaması 240/s 'a yakın istek alırken Monolitik uygulaması 180/s civarı istek alabilmiştir.



Şekil 2 HTTP Get Cevap Süreleri Grafiği

Kaynak: Gos ve Zabierowski (2020) ,2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)

Tablo 1 HTTP Get Cevap Süreleri Tablosu

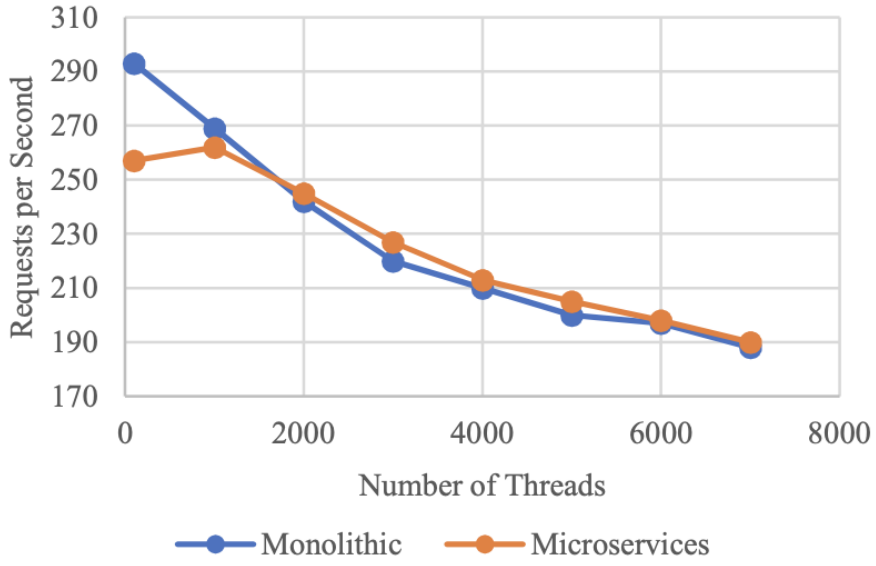
Architecture	30 000 req.	300 000 req.
Monolithic	12	90
Microservice (1 instance)	15	79
Microservice (2 instances)	16	72
Microservice (4 instances)	20	72

Kaynak: Gos ve Zabierowski (2020) ,2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)

Şekil ve tabloyu incelediğimizde 4 kez replika edilen varyasyon çok daha etkili olabileceği düşünülebilir. Ama bütün replika servisler aynı veritabanına bağlandığı için potansiyelini tam olarak gösteremediği görülmektedir.

Al-Debagy ve Martinek (2018) çalışmalarında Mikroservis mimarisi ile Monolitik mimarisini performans testi gerçekleştirmişlerdir. Bu testlerinde her iki mimarininde ne kadar sürede cevap döndüğü, ne kadar istek kabul ettiği test edilmiştir.

İlk senaryoda yük testi yapılmıştır. Bu testi yaparken 100 thread ile başlanmış ve bu sayı 7000 'e kadar çıkmıştır. Test sonuçlarına bakıldığında Mikroservis mimarisi ile Monolitik mimarinin birbirine yakın sonuçlar verdiği görülmüştür.

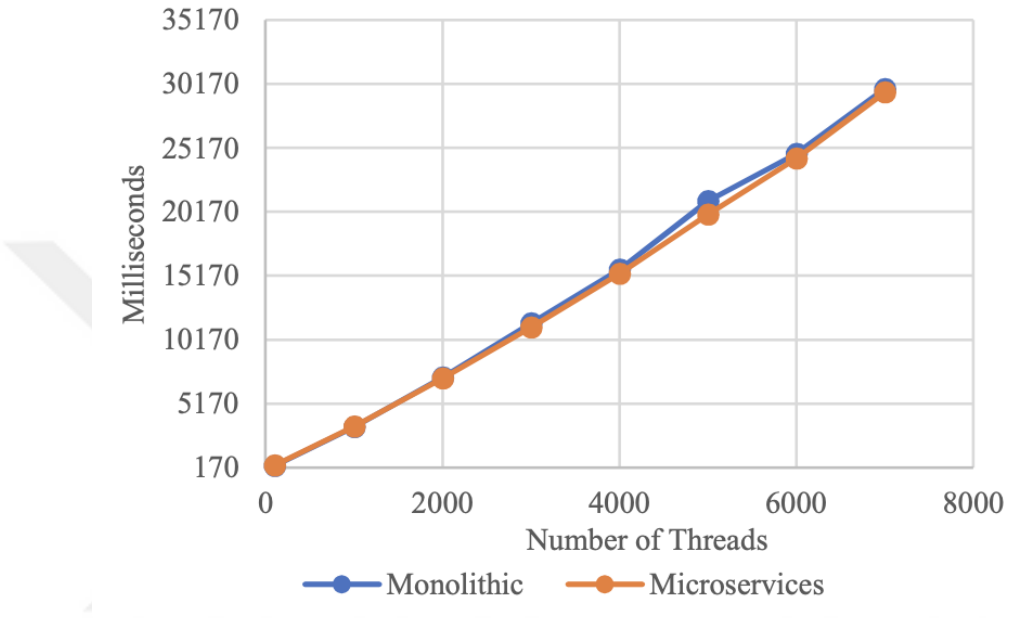


Şekil 3 Thread ve İstek sayıları Grafiği

Kaynak: Al-Debagy ve Martinek (2018) ,2018 A Comparative Review of Microservices and Monolithic Architectures. 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154.

100 thread ile işleme test işlemine başlandığında Monolitik mimarinin daha performanslı olduğu görülmüştür. Thread sayısı arttıkça her iki mimaride de performans düşüşü olduğu görülmüştür. Bu durumun oluşmasının nedeni, kullanıcı sayısı arttıkça isteklerin cevap süreleride artmaktadır. Bu durumda toplam çıktının reddedilmesine yol açmaktadır. Böylece Monolitik mimari 100 ile 1000 thread arasında daha fazla performans göstermersine rağmen 6000-7000 thread sayılarına ulaşıldığında Mikroservis mimarisi ile benzer performans sergilediği görülmüştür.

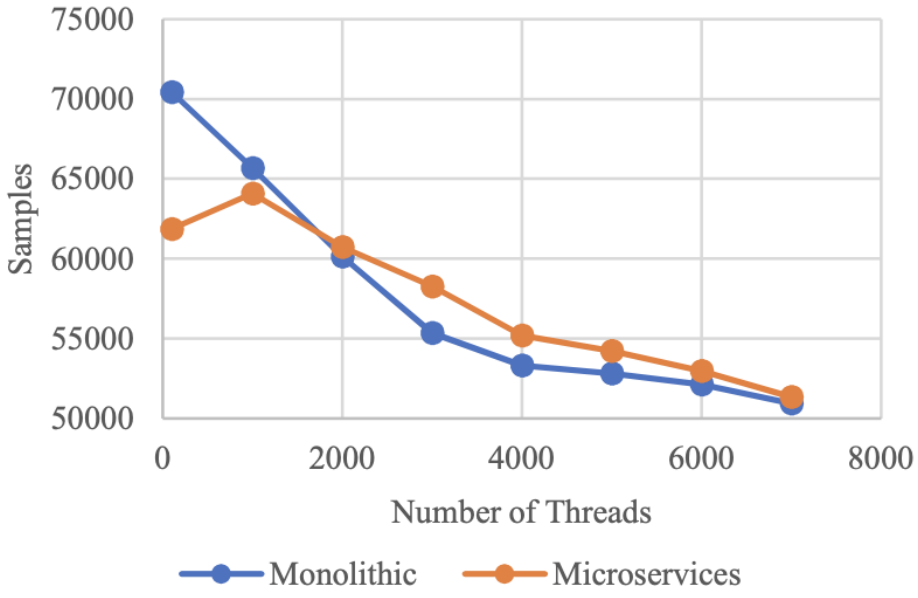
Bu senaryo temel alığındığında düşük sayılarıda kullanıcı ile Monolitik mimarinin daha performanslı olduğu görülebilir. Monolitik mimari ile Mikroservis mimarisi arasındaki performans farkı ortalama 0.87% olarak ölçülmüştür. Bu sonuca göre her iki yaklaşımda bu senaryo göz önüne alındığında belirgin bir performans farkı olmadığı söylenebilir.



Şekil 4 Cevap Sayıları Grafiği

Kaynak: Al-Debagy ve Martinek (2018) 2018 A Comparative Review of Microservices and Monolithic Architectures. 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154.

Elde edilen bir başka metrik ise cevap süreleridir. Cevap sürelerinde de büyük bir fark olmadığı görülmektedir. Thread sayısı arttıkça beklenildiği gibi cevap süreleride artmaktadır.

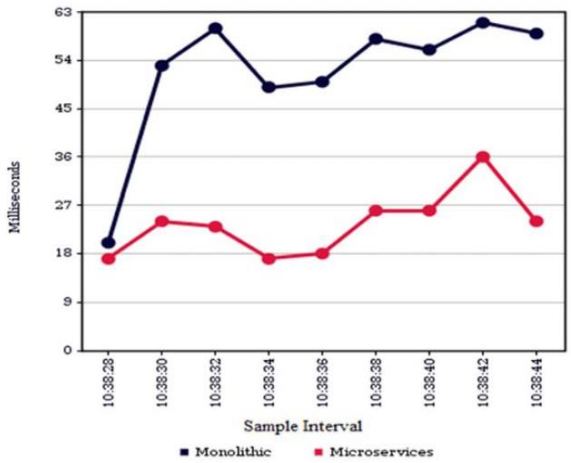


Şekil 5 Gerçekleştirilen İstekler Grafiği

Kaynak: Al-Debagy ve Martinek (2018) 2018 A Comparative Review of Microservices and Monolithic Architectures. 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154.

Bir başka metrik olarak her iki mimaride de gerçekleştirilen istek sayılarını ele alabiliriz. Bu sonuçlara göre düşük kullanıcı sayılarında Monolitik mimarinin daha performanslı olduğunu söyleyebiliriz. Kullanıcı sayısı arttıkça Mikroservis mimarisinin daha fazla performans sergilediği görülmektedir.

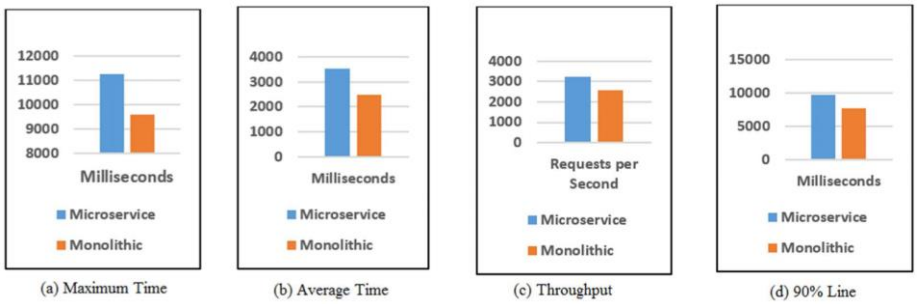
Singh ve Peddoju (2017) çalışmalarında Monolitik mimarisi ve Mikroservis mimarisini cevap ve yayınlama süreleri metriklerine göre bir karşılaştırma yapmışlardır. Yaptıkları çalışmada 50 kullanıcı 10 saniye boyunca istek göndermiştir.



Şekil 6 Mikroservis ve Monolotik Mimari Cevap Süreleri Grafiği

Kaynak: Singh ve Peddoju (2017) ,2017 Container-based microservice architecture for cloud applications. 2017 International Conference on Computing, Communication and Automation (ICCCA), 2017, pp. 847-852.

Düşük kullanıcı sayılarında Monolotik mimari daha yüksek performans göstermesine rağmen kullanıcı sayısı arttıkça Mikroserveri mimariye göre belirgin bir düşüş görülmüştür.



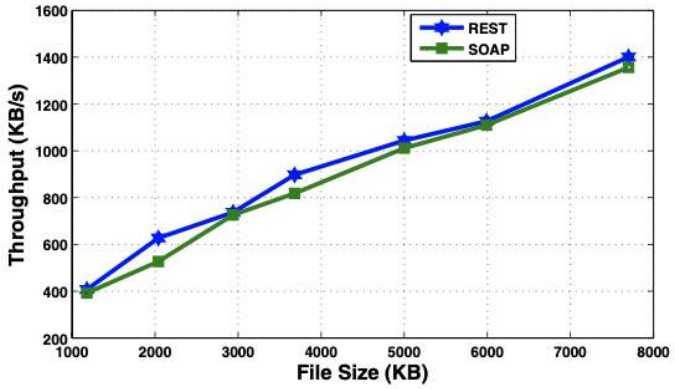
Şekil 7 Mikroservis ve Monolitik Mimari Yaklaşımının Performans Karşılaştırması

Kaynak: Singh ve Peddoju (2017) ,2017 Container-based microservice architecture for cloud applications. 2017 International Conference on Computing, Communication and Automation (ICCCA), 2017, pp. 847-852.

Yaptıkları bir başka testte ise kullanıcı sayısı 2000 ‘e çıkarılmış, 100 saniye boyunca 10 tekrar yapılarak istek gönderimi yapılmıştır.Bu testte Monolotik mimarinin çok daha düşük performans sergilediği görülmüştür. Bunun nedeni olarak Monolitik mimarinin, istekleri tek bir uygulama ile ele almasına rağmen Mikroservis mimarisi dağıtık bağımsız servislerce ele almıştır. Böyle Mikroservis mimarisi daha fazla çıktı üretebilmiştir.

Sonuç olarak Mikroservis mimarisinin Monolitik mimariye göre daha fazla performans gösterdiği görülmüştür. Ayrıca Mikroservis mimarisi ile geliştirilmiş uygulamaların daha yüksek ölçeklemeye olanak sağladığı belirtilmiştir.Mikroservis mimarisi bağımsız yapısını sayesinde farklı takımlar tarafından kurumsal uygulamalar geliştirilmesine ve otomatik yayınlama tasarımı sayesinde yayınlama süresini ciddi ölçüde düşürdüğü görülmüştür.

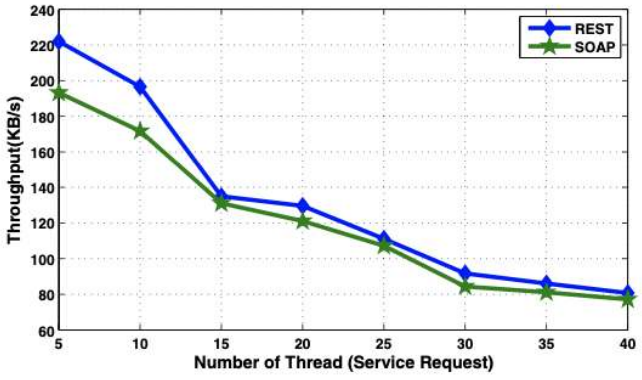
Kumari ve Rath (2015) çalışmalarında Mikroservis mimarisi ve Monolitik mimarilerini cevap süreleri, çıktı metriklerine göre kıyaslama yapmıştır. Bu kıyaslamayı yaparken 1000 kb ’tan 7000 kb ‘a kadar değişen dosyaların cevap ve çıktı süreleri gösterilmiştir.



Şekil 8 Dosya Büyüklüklerine göre Mikroservis ve Monolitik Kıyaslama Grafiği

Kaynak: Kumari ve Rath (2015) ,2015 Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration. 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2015, pp. 1656-1660.

Şekilde Mikroservis mimarisinin Monolitik mimarisinden daha iyi çıktı verdiği gösterilmiştir. Bunun nedeni olarak XML ile cevap vermesi olarak gösterilmiştir. Bu Mikroservis mimarisine göre daha yüksek dosya boyutlarına ulaşmasına neden olmaktadır.



Şekil 9 Farklı Servis İsteklerinin Çıktıları Grafiği

Kaynak: Kumari ve Rath (2015) 2015 Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration. 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2015, pp. 1656-1660.

Monolitik mimari ile Mikroservis mimarisinde istek sayılarına göre kıyaslandığında Mikroservis mimarisinin daha performanslı olduğu gösterilmiştir. Monolitik mimari istek sayısı biraz daha fazla olmasına rağmen cevap süresinin biraz daha düştüğü gösterilmiştir.

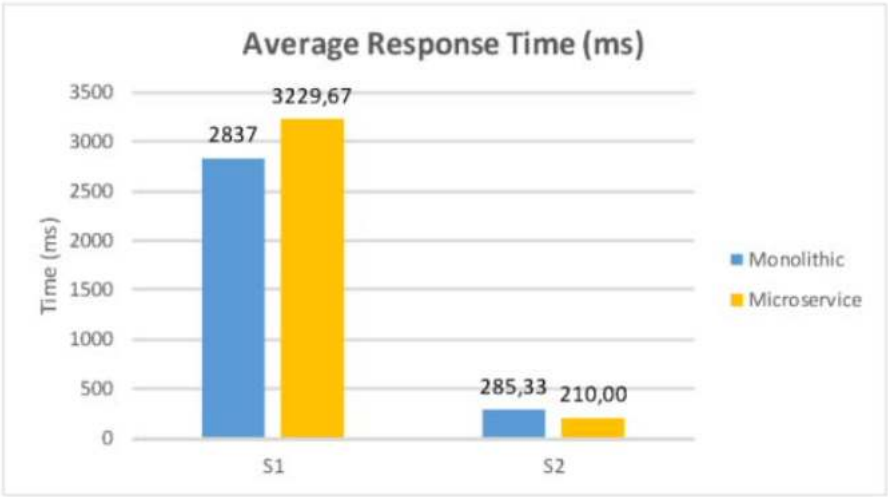
Sonuç olarak Mikroservis mimarisinin Monolotik mimariden daha performanslı olduğu gösterilmiştir. Ancak Monolitik mimari yeterli araç-gerece sahip olmasıyla oldukça popülerdir. Bu nedenle Mikroservis mimarisi ile geliştirme yapmak bir seçenek olarak tanımlanabildiği gösterilmiştir.

Villamizar , lar vd. (2015) çalışmalarında Monolitik mimari ve Mikroservis mimarisi ile geliştirmişlerdir. Geliştirdikleri uygulamaların iş gereksinimleri aşağıdaki gibi gösterilmiştir.

Tablo 2 Geliştirilen Her İki Servis için İş Gereksinimleri

Service	Requests per Minute	Average Response Time (ms)	Maximum Response Time (ms)
S_1 – Generate a payment plan	30	3,000	6,000
S_2 – Get a payment plan	1,100	300	1,500

Tabloda verilen iş gereksinimlerini karşılayan 2 ayrı Monolitik mimarisiyle ve 2 ayrı Mikroservis mimarisi ile geliştirme yapılmıştır. S_1 servisine dakikada 30 istek, S_2 servisine ise dakikada 1100 istek 10 dakika boyunca gönderilerek yük testine tabii tutulmuşlardır.



Şekil 10 Servislerin Yük Testinde Gösterdikleri Cevap Süreleri Grafiği

Kaynak: Villamizar , lar vd. (2015) 2015 Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. 2015 10th Computing Colombian Conference (10CCC), 2015, pp. 583-590.

Elde edilen sonuçlara göre Mikroservis mimarisinin sunucu maliyetlerinde 17% daha azaldığı görülmüştür. Monolitik mimari ile geliştirilen uygulamalar hızlı geliştirebilmeleri ve düşük kullanıcı sayılarında oldukça performanslı olmasına rağmen ölçekleme ve takım yönetimi gibi konularda problemlerle karşılaşıldığı görülmüştür. Mikroservis mimarisinin en büyük faydası olarak, binlerce veya milyonlarca kullanıcıya hitap eden büyük uygulamaların küçük uygulamalar halinde (Mikroservisler) yayınlanabilir, ölçeklenebilir ve yönetilebilir olduğundan bahsedilmiştir.

Yukarıdaki çalışmalardan da anlaşılacağı üzere bu tez kapsamında öngördüğümüz Monolitik mimariler, Mikroservis mimariler ve gRPC frameworkü kıyaslama çalışması literatürde yer almamaktadır.

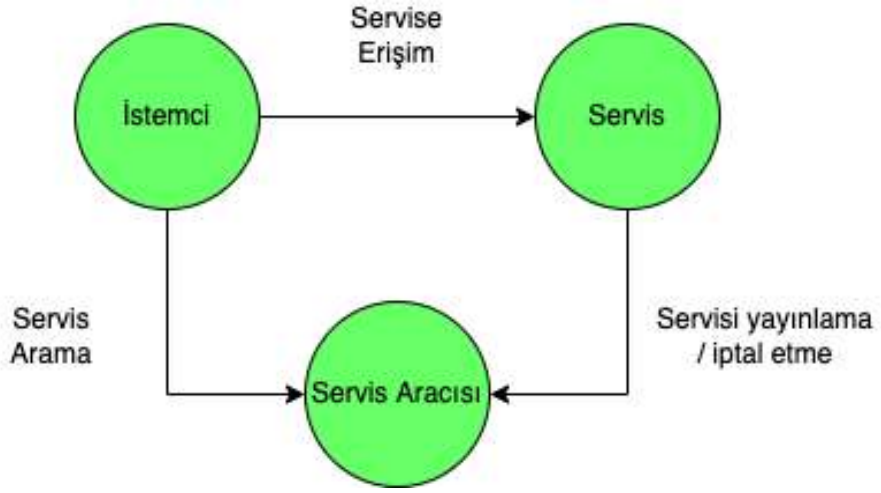
2. BU ÇALIŞMADA İNCELENEN MİMARİLER

2.1. Servis Mimarileri

Uygulamaların çeşitliliğine bağlı olarak geliştirme alanlarında aynı anda bir kaç farklı mimari kullanılabilir. Bu mimarilerin kullanımının seçiminde üzerindeki zaman baskısı, verilen termin süreleri ve değişiklik taleplerinin sıklığına göre tercih yapılır. Gelişen teknoloji ve değişen ihtiyaçlar neticesinde başarılı ve gelişime açık firmalar yeni teknoloji kullanımına odaklanmaktadırlar. Bunun sonucunda ihtiyaçlarına göre bir mimari seçerek yola devam etmektedirler.

Servis tabanlı mimari , kullanıcı gereksinimlerinin karşılandığı birbirine gevşek bağlı olarak çalışan yazılım parçacıklarının birbiri ile etkileşimde bulundukları bir mimaridir. (He 2003) Yazılım geliştiriciler, uygulamadan beklenen fonksiyonaliteye uygun şekilde servis kolleksiyonları arasından ihtiyaç olan servisleri seçerek yazılım geliştirirler.

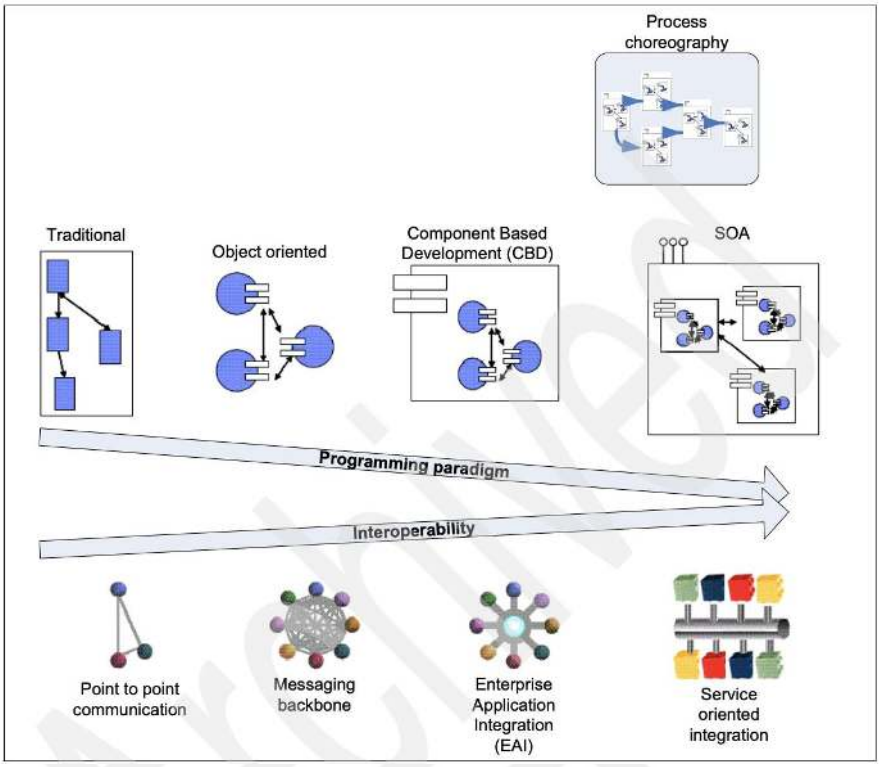
Servis tabanlı mimari, öncelikli olarak dağıtık sistem dizaynı ve geliştirmesi için kullanılır. Oldukça esnek geliştirme olanakları sayesinde farklı görevleri icra edebilir, gevşek bağlı olma özelliği sayesinde tekrar kullanılabilir olmaktadır. Ayrıca farklı sistemlerinde bağlanmasına olanak sağlayarak aynı şekilde kullanım imkanı sunmaktadır. Servis belirli bir sistem fonksiyonunu gerçekleştiren bir yazılım bileşenidir ve servis sağlayıcı tarafından kullanıcıların erişimine açıktır (Ramanathan ve Korte 2014) .



Şekil 11 Servis Odaklı Mimari

Servis tabanlı mimaride üç farklı rol mevcuttur. Bunlar servis talep eden, servis sağlayan, servis aracısıdır. Servis tabanlı mimari platform bağımsız, tekrar kullanılabilirlik, gevşek bağlı ve diğer servisler ile birlikte çalışabilirlik sağlayan bir teknolojidir. Bu teknolojinin ana avantajı birbiri ile etkileşime girebilmesi için ortak bir yolu desteklemesidir. Gerektiğinde kaldırılıp yerine yeni servisler eklenebilir ve yazılımın dinamik kalması sağlanabilmaktedir (Gehlen ve Pham, 2005).

Servis tabanlı mimariler kurumsal yapılarda rollere göre farklı anlamlar içerebilir. Yönetici seviyesindekiler ve iş analistleri için iş ortakları ve müşterilerin kullanabileceği bir bilişim teknolojisinden ibaret olabilir. Kurumsal bir yazılım mimari için modüler, yeniden kullanılabilir, gevşek bağlı, birleştirilebilir bir mimari displin veya dizayn çözümleri olarak görülebilmektedir. Bir proje yöneticisi veya testçi için ise farklı anlamlara gelebilir. Servis odaklı mimari her rol için farklı anlamlara gelebilmektedir. Bu anlamların hiç biri yanlış olduğu söylenemez (Darmawan , lar vd., 2008).



Şekil 12 Servis Odaklı Mimariye Doğru Değişim

Kaynak: Darmawan , lar vd. (2008) ,2008 Power Systems and SOA Synergy.

Yazılım geliştirme paradigması metodsal bakış açısından nesneye yönelik yazılım geliştirmeye doğru gelişim göstermiştir. Sonrasında bileşen tabanlı mimari ve son olarak servis tabanlı mimariye doğru bir gelişim göstermiştir. Bu değişim ve gelişim uygulamaların birbirleri arasındaki iletişimi artırmıştır. Geleneksel uygulamalar birbirleri ile iletişim kurmaları için bir mekanizmaya sahiptirler. Ama bunu genişletmek oldukça zordur. Nesneye yönelik yazılım geliştirme ve bileşen tabanlı yazılım geliştirme potansiyel olarak esnek bir iletişim yapısına sahiptir. Servcis odaklı mimari bu yazılım geliştirme mekanizmalarını kullanarak kurumsal uygulama geliştirme alt yapısı geliştirilebilir. (Darmawan , lar vd., 2008).

Teknolojinin gelişmesiyle servis odaklı mimaride yöntem ve yaklaşımda gelişmeye başlamıştır. Bu gelişme sonrasında ortaya çıkacak olan Mikroservisler ile devam etmiştir.

2.1.1. Monolitik Servisler

Kurumsal seviyede geliştirme yapılan sistemler geleneksel olarak monolitik olarak dizayn edilir. Monolitik uygulamalar bağımsız olarak çalıştırılamayan uygulama bölümleri olarak tanımlanabilir. Bu mimaride artan karmaşıklıkla birlikte ölçekleme ve bakım yapılması giderek zorlaşır. Bunun sonucu olarak harcanan zaman giderek artar.

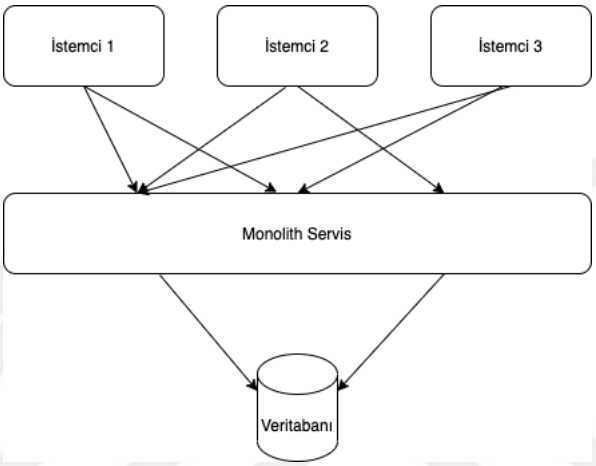
Modüler olarak geliştirilebilir olmasına rağmen uygulama monolitik olarak yayınlanır. Bu format uygulamanın diline ve işletim sistemine bağlı olarak çalışır. Örnek olarak çoğu java uygulaması WAR formatında paketlenir ve sunucuda Tomcat veya Jetty uygulamalarında yayınlanır. Tek bir uygulama geliştirmeye odaklanıldığında IDE veya diğer araçlar ile basit bir şekilde geliştirme yapılabilir. Bu tür uygulamalar kolaylıkla test edilebilir. Uygulamayı basitçe çalıştırarak baştan sona Selenium gibi araçlarla test edilebilir. Monolitik uygulamaları yayınlaması da oldukça basittir. Paketlenmiş uygulamayı sunucuya kopyalamak yeterlidir. Birden çok kopyasını yük dengeleyici yardımıyla çalıştırarak ölçekleyebilirsiniz. Projenin ilk aşamalarında oldukça iyi çalışabilir (Richardson , S., 2016).

Ancak bu basit yaklaşımın büyük bir sınırlaması vardır. Bu tür başarılı uygulamalar da geliştirme arttıkça daha büyük paketler halini almaya başlar. Geliştiriciler yeni özellikler eklemek için uygulama üzerinde çalıştıkça birkaç yıl içinde bu küçük uygulama büyük bir monolitik uygulama halini alır. Uygulama büyüdükçe kompleks bir hal alır ve geliştirici takımda baş ağrısına neden olabilir. Tek bir geliştirici için bile uygulamayı anlaması da ayrı bir sorun yaratır. Sonuç olarak uygulamadaki hataları gidermek ve yeni özellikler eklemek oldukça zorlaşır. Bu da geliştirme süresini artırarak uygulamanın ayağa kalkma süresini yükseltir (Richardson , S., 2016).

Başka bir problem ise bu büyük kompleks monolitik uygulamanın sürekli geliştirme yapılmasına bir engel oluşturmasıdır. Günümüzde SaaS uygulamaların üretim ortamına gün içinde bir çok sefer güncelleme gönderilir. Bu monolitik bir uygulama için oldukça karmaşıklaşır. Uygulamanın bir parçasını güncellemek için bütün uygulamayı tekrar yayınlamanız gerekir. Uygulamanın ayağa kalkması ve

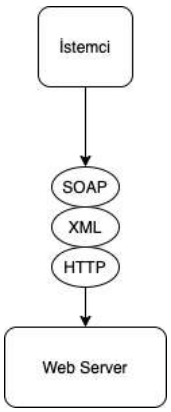
geliştrilen yeni özelliklerinin çalışıp çalışmadığını anlamak el ile test yaparmış gibi bir hal alır. Sonuç olarak sürekli geliştirme imkansızlaşır (Richardson , S., 2016).

Monolitik uygulamaların modüllerinde ortaya çıkan kaynak çakışması da ölçekleme işlemini zorlaştırır. Örnek olarak uygulamanın bir bölümü işlemci yoğun bir işlem yaparken başka bir bölümü ise bellek yoğun bir işlem yapabilir. Ancak bu modüller birlikte yayınlanır. Bu nedenle verimli çalışması için gerekli donanım seçimi de zorlaşır.



Şekil 13 Uygulama Ve Servis Etkileşimi

Geliştiricilerin monolitik uygulamanın ne yaptığını ve nasıl çalıştığını anlamaları uzun sürebilir. Varolan bağımlılıkları çözmeleri ve yeni bağımlılıklar eklemeleri oldukça zahmetli bir süreç gerektirir. Bir bağımlılığı güncellerken bu bağımlılığın uygulamanın başka hangi parçalarında sorunlara yol açacağını önceden kestirmek zorlaşır (Raj ve Ravichandra 2018).



Şekil 14 SOA Diagram

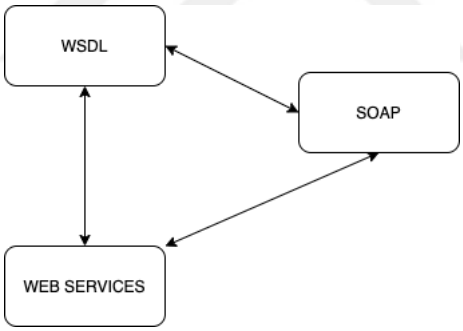
İstemciler, WSDL McGovern , Tyagi , Stevens, ve Mathew (2003) dökümanına bakarak servisin sahip olduğu metodları ve bu metodların talep ettiği veri tipleri ve cevap olarak dönecekleri veri tiplerini istemcilere iletirler. Böylece istemciler ne tipte veri alacağını bilerek çalışmalarını yürütürler.

2.1.2. Monolitik Servislerin Faydalı Yanları

Araştırmasında da belirtildiği gibi monolitik uygulamaların aşağıdaki gibi faydaları mevcuttur (Raj ve Ravichandra 2018) .

- **Servis Kontratı**

Servis kontratı, WSDL olarak tanımlanan bir döküman ihtiva eder, bu döküman yardımıyla istemciler servis ile nasıl etkileşimde bulunulacağını öğrenirler. Böylelikle istemci sunucuda hangi metodları ve istek veya dönüş yapılan veri tiplerini öğrenebilir. Bu kesinlik sayesinde geliştirici istenen paremetreleri ve dönüşleri kolaylıka yönetebilir.



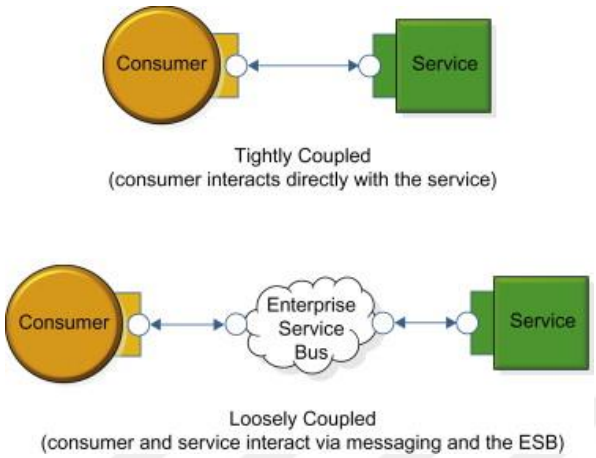
Şekil 15 WSDL Kullanım Şeması

- **Gevşek bağlılık**

Servis, istemciler arasında yürütülen arka plan işlemleri birbirleri ile paylaşılmaz ve gizli tutar. Böylelikle bağımlılık çok düşü seviyede kalır, istemci yada servis güncellemeleri bu nedenle birbirlerini etkilemez.

Servisler ve istemcier bu özelliklerini korumak için birbirleri arasında fiziksel bağımlıklarının olmasından kaçınırlar. Bu özelliğin tam olarak uygulanabilmesi için

mesajlaşma ile iletişim kurmaları beklenir. Böyle servis ve istemci doğrudan bağlantı kurmazlar. Servis tarafında geliştirilen ilave işlevsellikler istemci tarafını doğrudan etkilemez (Bean 2010).



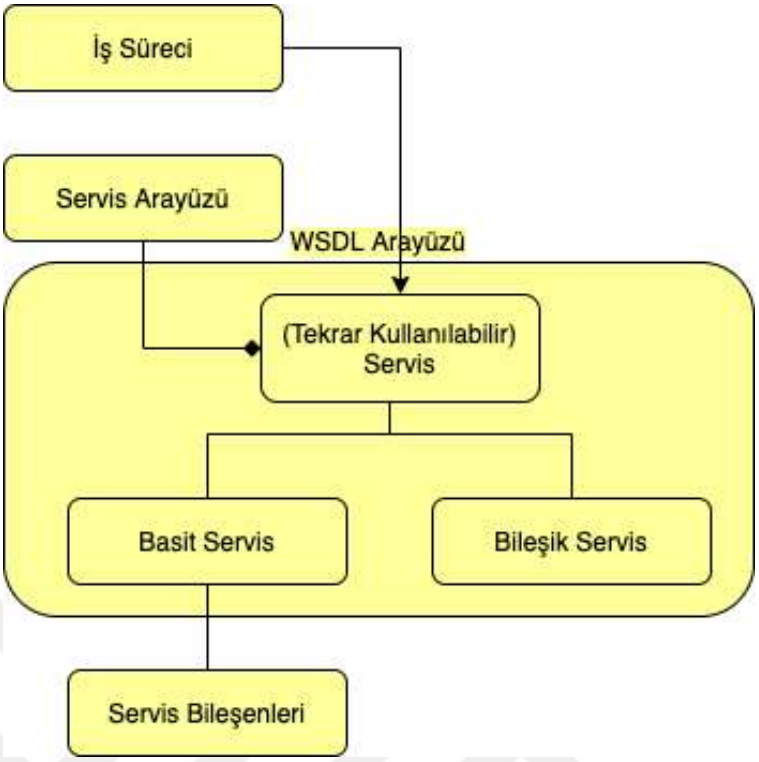
Şekil 16 SOA Gevşek Bağlı Yapı

Kaynak : Bean (2010) ,2010 Core SOA Principles. *In* SOA and Web Services Interface Design. J. Bean, ed. Pp. 25-41. Boston: Morgan Kaufmann.

- **Tekrar Kullanılabilirlik**

Servisler farklı istemciler veya servisler tarafından tekrar kullanılabilir, böylece her hangi bir iş gereksiniminden bağımsız olarak servisler işlerini icra edebilirler.

Yeni istemcilerin ihtiyaçları varolan servisler ile karşılanabiliyor ise geliştiriciler yeni servis geliştirmeye ihtiyaç duymazlar ve varolan servisleri kullanabilirler. Birde varolan servislere eklenen yeni işlevsellikler daha önce geliştirilmiş uygulamalar için de kullanılabilir olmaktadır veya yeni istemcilerin gerek duyduğu yeni işlevsellikler varolan servislere eklenerek yeni servis geliştirilmesi gerekliliği ortadan kalkmaktadır. Böylece düşük teknik borçlanma ile var olan uygulamalarda veya yeni uygulamalarda güncelleme yapmak kolaylaşır. Bu servislerin en güçlü yanlarından birisidir, varolan istemciler etkilenmeden servis arayüzleri değiştirilebilir ve bu durum varolan istemcilerden gizlenebilir.(Bean 2010)



Şekil 17 Servis Tekrar Kullanılabilirliği

- **Servis Durumsuzluğu**

Servisler durumsuz olarak çalışırlar böylece birden fazla isteği işleyerek hızlı dönüş yapabilirler. Bu özellik aynı zamanda servis performansını artırır, ölçekleme ve yeniden kullanım işlemlerinde kolaylık sağlar. Böylelikle kaynakların aşırı kullanımından oluşan sistem kilitlenmelerinden kaçınılır (Darmawan , lar vd., 2008).

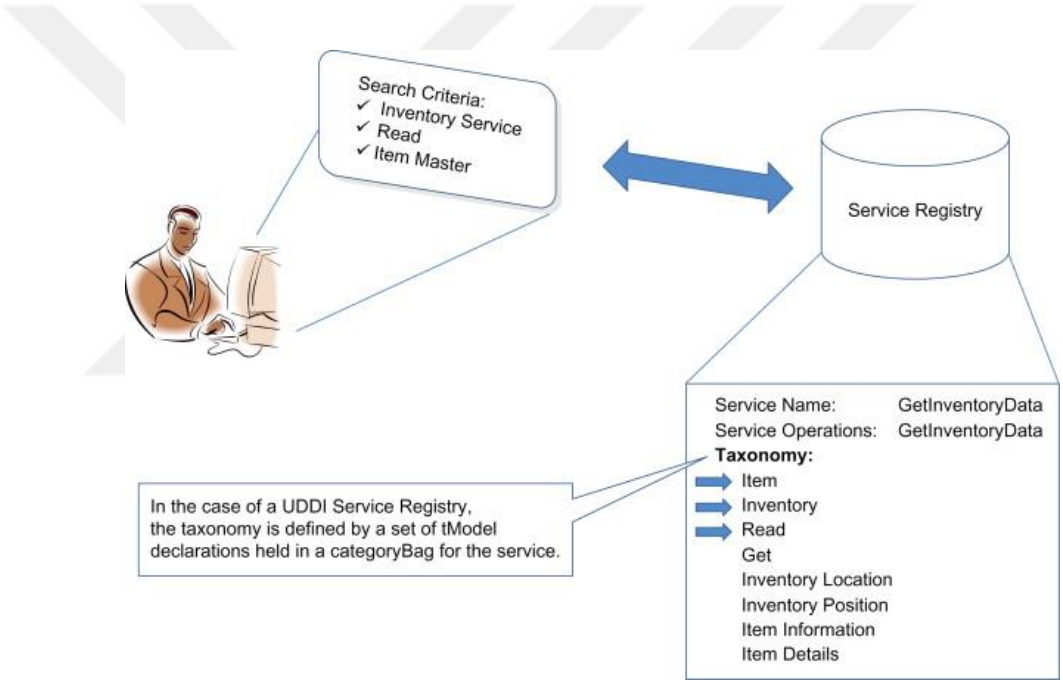
- **Servis Birlikte Çalışabilirliği**

Servisler birbirleri ile veri paylaşabilirler, eğer veri paylaşımına uygun değilse paylaşılan veriye uygun çalışma yapılması gerekebilir. Ayrıca birbirleri ile etkileşimde bulunan servisler tutarlı veri tipleri ve biçimleri içermesi nedeniyle birlikte çalışarak yeni bir servis oluşturulabilirler (Darmawan , lar vd., 2008).

- **Servis Bulunabilirliği**

Servis, yayınlandığı ortamda intranet, internet kolaylıkla erişilebilir ve servis kontratı ile kolaylıkla iletişim kurulabilir.

Geliştiriciler, daha önce geliştirilmiş servisleri Servis Kayıt Sisteminden sorgulayarak bulabilirler. Servis kayıt sistemi yayınlanmış servisleri bir katalog veya depo şeklinde ihtiva eder. İstenilen servisin bulunmasını takriben geliştiriler istemcilerin ihtiyacını karşılayıp karşılamayacağını değerlendirir. Eğer ihtiyacı karşılıyor ise yeniden kullanılabilir, ihtiyacı karşılamıyor ise varolan servise ihtiyacı uygun yeni işlevsellikler katılabilir.



Şekil 18 SOA Servislerinin kayıt sistemi ile yeniden bulunabilirliği

Kaynak: Bean (2010) ,2010 Core SOA Principles. *In* SOA and Web Services Interface Design. J. Bean, ed. Pp. 25-41. Boston: Morgan Kaufmann.

2.1.3. Monolitik Servislerin Olumsuz Yanları

- **Birlikte Çalışılabilirlik**

Birlikte çalışabilirlik bazı ek çalışmalara gereksinim duyar. Farklı dil veya ortamlarda geliştirilen servisler birbirleri ile iletişim kurmaları zorlaşabilir. Bu da karmaşaya neden olarak bakım ve geliştirme maliyetlerini artırabilir.

SOA servisleri farklı teknolojiler kullanılarak geliştirilebilir. Örneğin bir teknoloji java ile geliştirilmiş ve Linux sistem üzerinde çalışıyorken, bir başka servis Visual C++ veya C# ile geliştirilmiş olabilir. SOA yapısı bu servislerin farklı teknolojiler ile geliştirilmilş olmasına rağmen birlikte çalışmasına olanak verir. Bunu sağlayan özellik ise neredeyse bütün platformların kabul ettiği XML veri yapısıdır. XML yapısını destekleyen bütün platformlar XML'i okuyabilir veya bir XML verisi oluşturabilir (Bean 2010).

- **Ölçeklenebilirlik**

Geliştiriciler yeni bir servis oluşturarak bunları istemcilere hızlıca açabilirler. Açılan bu servisler istemcilerin ihtiyacına göre yeterince ölçekleme yapabilir olmalılar.Ölçekleme sistem performasının artırılması üzerine yapılan bir çalışmadır. Varolan donanım yapısınında buna uygun olması beklenir. Bu işlemler sanallaştırma, sanal sunucu oluşturma veya yük dengeleyici gibi seçenekler ile gerçekleştirilebilir (Darmawan , lar vd., 2008).

Monolitik uygulamalar, ölçekleme işlemleri çin bir engel oluşturur. Uygulamanın birden fazla kopyası çalıştırılarak ölçekleme yapılabilir. Ancak bu mimari de veri işleme ölçeklenemez. Çünkü bütün kopyalar verinin tümüne erişebilir. Bu tamponlamayı daha az efektif kılar, bellek kullanımını artırır, giriş-çıkış trafiğini artırır ve uygulamanın farklı bileşenleri farklı kaynaklara erişim ihtiyacı duyabilir. Bir bileşen işlemci gücü içeren işlem yaparken bir başka bileşen yüksek bellek gereken bir işlem yapabilir. Monolitik uygulamalarda bileşenler ayrı ayrı ölçeklenemez (Richardson , C., 2021b) .

- **Eşgüdümlülük**

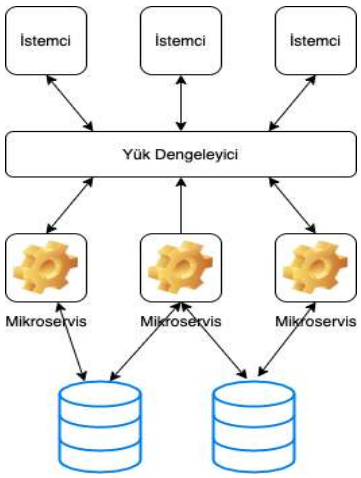
Servisler arasındaki etkileşim noktadan noktaya sağlanır. Buda zamanla hangi servisin hangi yolu izleyeceği karmaşasına neden olur. Geliştiriciler yaptıkları geliştirme ve değişiklikleri uygulamada zorluk yaşayabilirler.

- **Uzun Soluklu Aynı Teknoloji kullanımı**

Monolitik mimari geliştirme takımını geliştirmenin başında kullanılan teknolojiye ve hatta kullanılan kütühanenin belirli bir versiyonuna bağımlı kılar. Yeni teknolojiye adapte olma süüreci oldukça zorlaşır. Örneğin .net üzerinde bir uygulama geliştirdiğimizi varsayalım. Bu uygulamanın ileriki versiyonlarında farklı bir .net sürümü veya altyapı kullanmamız zorlaşabilir. Bu işlemi yapabilmek için neredeyse bütün uygulamayı yeniden yazmak gerekebilir. Bu durum oldukça riskli bir hal alır (Richardson , C., 2021b).

2.1.4. Mikroservisler

Mikroservisler, ölçeklenebilir, bakım maliyeti çok daha düşük ve birbirinden bağımsız olarak yayınlanabilir uygulamalar olarak ön plana çıkmaktadır. Böylelikle çalışma alanı özellikli küçük yayınlanabilir uygulamalar geliştirilerek yeni bir mimari yaklaşım olarak ön plana çıkmıştır.

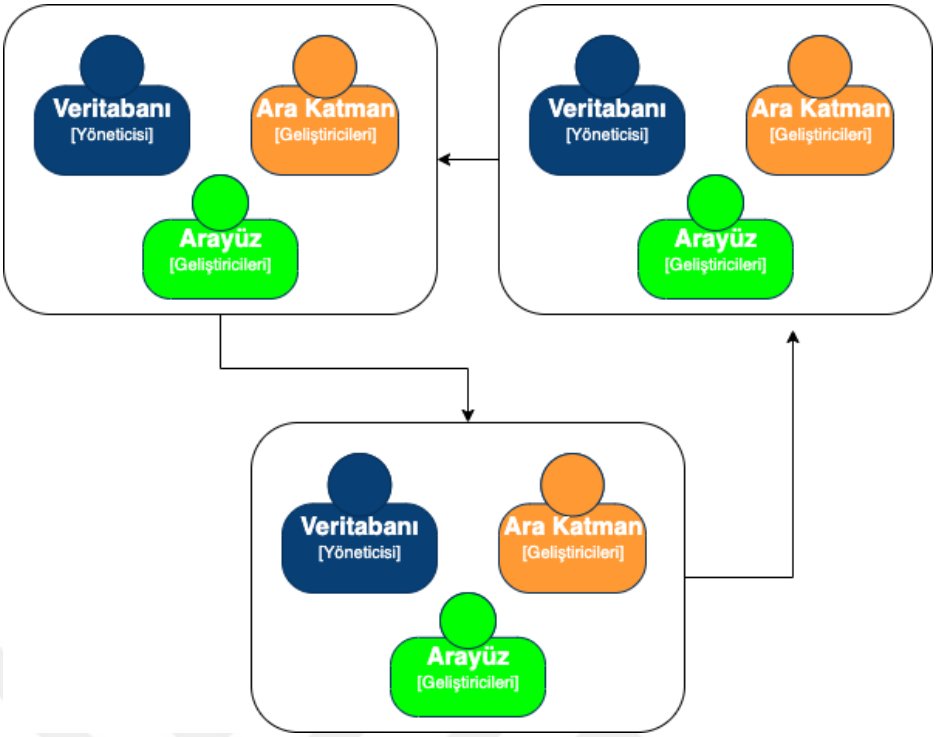


Şekil 19 Mikroservis Uygulama Şeması

Yenilikçi ve gelişime açık firmalar artan mobil kullanıcı sayısınıda göze alarak anlık veri miktarının artması, artan sunucu maliyetleri ile birlikte sunucuların etkin kullanımı gerektirmesiyle monolitik yapıları mikroservis mimarisine dönüştürmeye başlamışlardır (Murugesan, 2007).

Mikroservisler, bağımsız bir şekilde yayınlanabilir, bağımsız bir şekilde ölçeklenebilir ve bağımsız bir şekilde test edilebilir küçük uygulamalardır. Bir işi yapmaktan sorumlu olabilirler böylece kolaylıkla değiştirilebilirler ve anlaşılır olabilirler. Gün geçtikte daha çok geliştirici tarafından kullanılmasının nedenlerinden birisi de sürekli teslimat yapısına uygun olması ve kolay ölçeklenebilir olması olarak gösterilebilir. Geliştiricilere çeşitli kolaylıklar sağlamaktadır. Örneğin çok kullanılan diler olan C# veya Java ile geliştirilebilir. İhtiyaca göre farklı araç-gereçler kullanılarak geliştiriciye özgür bir ortam sunar (Thönes 2015).

Http kaynaklarını kullananarak iletişim kuran hafif uygulamalar olan mikroservisler, bileşenlere ayrılmış servisler olarak da görebiliriz. Bu bileşenleri bir kütüphane olarak tanımlayıp istemci tarafındaki uygulamanın fonksiyonları ile çağırabiliriz. Bir önemli kriter de mikroservisler bağımsız olarak yayınlanabildikleri için bir bileşen olarak tanımlanabilir. Eğer bir uygulama bir işlem için birden fazla kütüphane içeriyorsa bu kütüphanelerden birinin değişimi bütün uygulamanın yeniden yayınlanmasına sebep olur. Uygulama birden fazla servise ayrılmış olsaydı. Sadece değişiklik gerektiren servis yeniden yayınlanarak işlem tamamlanmış olurdu. Böylece mikroservis mimarisi bağlı kaldığı sınırları ve servis kontroları mekanizmasını minimize etmiş olur (Levis, 2014).



Şekil 20Mikroservisler Çapraz Bağlı Takım Çalışması

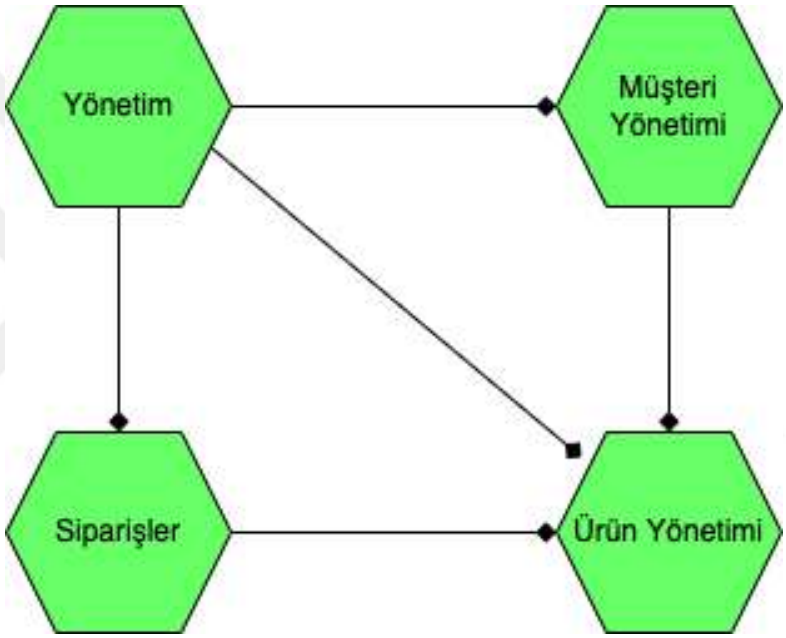
Mikroservis yaklaşımı, takımları iş kapasitesine göre ayırır. Bütün takım üyeleri önyüz, arayüz ve veritabanı kısmında birbiri ile etkileşim halide çalışırlar. Böylelikle her bileşen kendi içinde çalışmasını sürdürerek bağımsız olur ve üretim ortamında geliştirilen projenin nasıl davranış gösterdeği ile ilgili günden güne izleme olanağına kavuşurlar. Bu sayede geliştiriciler uygulamalarına fonksiyonlar bütünü olarak bakmak yerine iş kapasitesine odaklanırlar (Levis, 2014).

Bu tür servisler farklı özellik veya fonksiyonaliteyi içerebilir. Örneğin bir e-ticaret uygulamasından bahsedecek olursak, bunlar yönetim, müşteri yönetimi, sipariş yönetimi, ürün yönetimi vb. olabilir. Her mikroservis kendi iş mantığını gerçekletiren küçük bir uygulamadır. Bazı mikrosevislere bir API ile erişilebilir, böylelikle başka mikroservisler tarafından erişilebilir olurlar. Bu mikroservisler bir web önyüzü içerebilir (Richardson , S., 2016).

2.1.5. Ne zaman Mikroservis Mimariyi Kullanmalıyız

Uygulama geliştirmeye başlarken bir problem ile karşılaşmadığınız için hangi mimari yaklaşım içerisinde olmanız gerektiğinin önemi olmayabilir. Detaylı ve

dağıtık bir mimari geliştirme hızınızı yavaşlatabilir. Bu durum başlangıç aşamasında bir sorun olarak görülebilir. Geliştirme takımı uygulamayı olabildiğince hızlı tamamlamak isteyebilir. Sonrasında ise ölçekleme yapma, fonksiyonel ayrıştırma ve karmaşık bağımlılıklar monolitik uygulamayı servislere ayırma işlemini zorlaştırabilir. Geliştirilen uygulama büyük ölçekli olmayacaksa ve sadece belirli fonksiyonları içerecekse monolitik mimari ile devam edilebilir. Ancak uygulama farklı fonksiyonları içereceği öngörülüyorsa ve ileri ki zamanlar ölçekleme ihtiyacı duyulacak ise mikroservis mimarisi devam etmek daha faydalı olabilir (Richardson , C., 2021a).



Şekil 21 E-Ticaret Sisteminin Mikroservis Mimarisi ile Uygulanması

Şekilde her mikroservis bir hexagon olarak gösterilmiştir. Bu mikroservisler diğer mikroservislerin bağlanabilmesi için bir API yayınlamaktadırlar. Yönetim servisi müşteriler ile ilgili işlem yapabilir, siparişleri kontrol edebilir veya ürün yönetimi ile ilgili işlemleri yapabilir. Siparişler servisi ise verilen siparişin hangi ürüne ait olduğunu ürün servisinden alabilir ve hangi müşterinin siparişi verdiğini müşteri yönetimi servisinden elde edebilir.

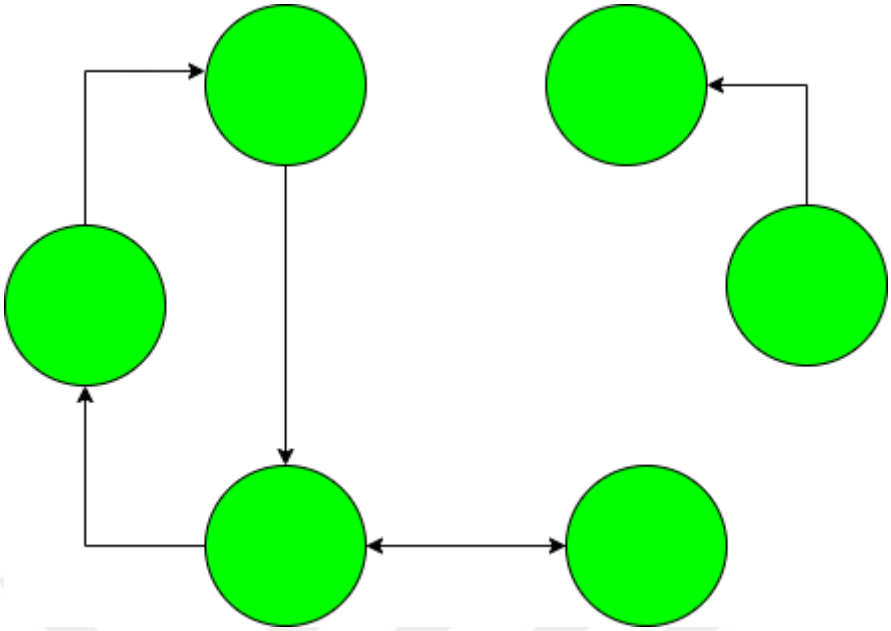
2.1.6. Mikroservislerin Avantajları

Mikroservis mimari dizaynının faydalarından bahsederken ilk önce karmaşıklık problemi ile başa çıkabilmenin etkili bir yolu olarak belirtilebilir. Monolitik olarak yazılmış bir uygulamayı servisler kümesine dönüştürülebilir. Bununla birlikte uygulamanın fonksiyonalitesinde bir değişiklik olmaz. Her servisin kesin belirlenmiş sınırları olabilir ve bir monolitik uygulamada başarılması zor olan modülariteye tamamiyle imkan sunar. Sonuç olarak bağımsız servisler hızlı geliştirilir, hızlı anlaşılır ve bakım yapılır. Her bir takım birbirinden bağımsız çalışabilir ve yayınlama yapabilir. Takımlar kullanacakları teknolojileri belirlemekte özgürdürler. Tabii kurumlar yeni teknoloji seçimlerinde seçenekleri sınırlayabilirler. Netice itibariyle geliştiriciler bu özgürlüklerini eski teknolojileri terk edip yeni teknolojileri devreye alarak çalışmalarına devam ettirebilirler. Servis yazılmaya başlandığında kullanılan teknoloji eski bir teknoloji olabilir. Ama ilerleyen süreçte yeni geliştirilen servisler de yeni teknolojiler kullanılabilir. Bunun yanında küçük boyutlarda olan servislerin yeni versiyonu yazılmaya başlandığında bu süreç çok uzun olmayacaktır (Richardson , S., 2016).

Araştırmasında da belirtildiği gibi mikroservislerin aşağıdaki gibi faydaları mevcuttur (Raj ve Ravichandra 2018) .

- **Gevşek Bağlılık**

Mikroservisler tam olarak gevşek bağlı olarak çalışabilirler. Her mikroservis kendi veritabanını kullanabilir, böylece veritabanı üzerinden veri paylaşımı yapmak zordunda kalmazlar. Bu da benzer kodların farklı servislerde kullanımını azaltır.



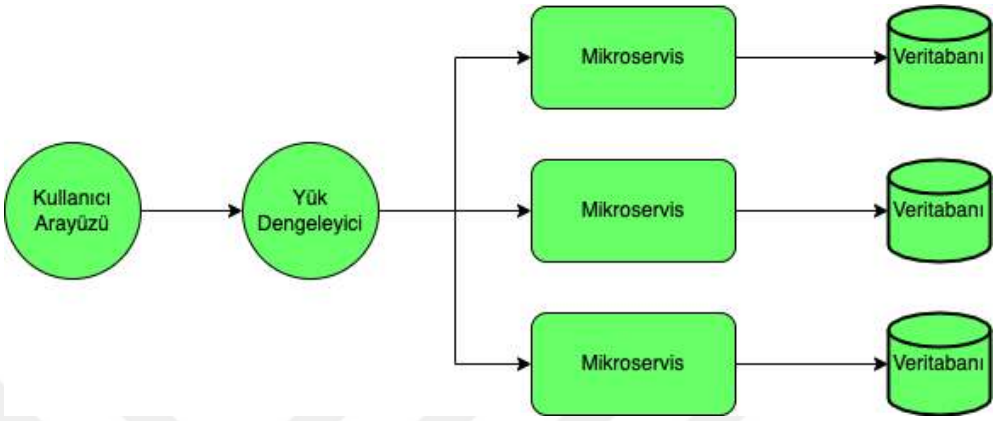
Şekil 22 Mikroservisler Gevşek Bağlı Çalışması

- **Ölçekleme**

Servisin bazı bölümlerini ölçekleme imkanı verdiği için bütün sistemi ölçekleme ihtiyacı olmaz böylece performans sorunları ile karşılaşma oranı azalır.

Şekil 23 Mikroservis Dikey Ölçekleme

Mikroservis sunucularında donanımsal yükseltmeye gitmek dikey ölçekleme olarak kabul edilir. Sistemin işlemci gücü, bellek kapasitesi, disk kapasitesi, ağ bağlantısının artırılması daha güçlü sunucu elde edilerek performans artışı sağlanabilir.



Şekil 24 Mikroservis Yatay Ölçekleme

Sunucuda istek bekleyen servis sayısı artırılabilir. Bu servisleri de yük dengeleyici sayesinde eşit yük dağıtacak şekilde yapılandırabiliriz. Böylelikle sunucu maliyetinde bir değişiklik olmadan performans artışı sağlanabilir.

- **Durumsuzluk**

Mikroservisler durumsuzdurlar. Herhangi bir durum bilgisi tutmadıkları için gelen isteğe gerçekleştirmesi gereken işlev ne ise gerçekleştirip dönüş yaparlar. Böylece ölçeklenebilirlik artar. SOAP'ta olduğu gibi durum kontrolü olmadığı için aynı sistemde çalışma zorunlulukları yoktur.

Bunun yanında , bakım maliyetinin düşmesi ve servisin küçük boyutlarda olması nedeniyle değiştirme ve anlama süresi oldukça kısalmaktadır. Servisi test etmek çok daha kolay olur. Servis diğer servislerden bağımsız olarak yayınlanabilir. Her geliştirme takımı sorumlu olduğu servisleri diğer servislerden bağımsız olarak çalıştırabilir ve yayınlayabilir böyle takımların işlerini organize etmesi dahada kolaylaşır (Richardson , C., 2021a).

- **Çeviklik**

Mikroservisler bağımsız olarak yayınlanabilir. Hataları düzeltme ve yeni özellikleri eklemesi çok daha kolaydır. Bütün bir uygulamayı güncellemeden sadece servisi güncelleyebilirsiniz veya hata oluştuğunda bir önceki versiyona kolayca dönebilirsiniz. Bir hata oluştuğunda bu uygulamanın geri kalanını etkileyebilir, bu nedenle yeni özellikleri yayınlarken teste tabi tutulduktan sonra yayınlanması gerekir (Microsoft, 2021b).

- **Küçük Takımlar**

Küçük takımların sorumlu oldukları kod blokları küçük olduğundan derleme, test ve yayınlama işlemleride daha kolaylaşır. Buda takıma çeviklik katar. Büyük takımlarda iletişim daha az olduğu için daha az üretken olmaya eğilim gösterirler (Microsoft, 2021b).

- **Temel Küçük Kod Parçaları**

Monolitik uygulamada bağımlılıklar zaman içinde aşırı karmaşıklaşmaya eğilim gösterir. Yeni bir özellik eklemek, kodun bir çok yerine dokunmak anlamına gelebilir. Mikroservis mimarisi, veritabanı ve kod paylaşımını en aza indirdiği için yeni özellik eklemek çok daha kolay olur (Microsoft, 2021b).

- **Yeni Teknolojileri Birleştirme**

Takımlar yeni teknolojileri, geliştirdikleri servise en uygun şekilde entegre edebilirler (Microsoft, 2021b).

- **Hata İzolasyonu**

Eğer bir mikroservis erişilemez olursa, bu durum bütün uygulamada kesintiye uğratmaz. Tabii bunun için mikroservisin oluşan hataları uygun bir şekilde ele alabilecek şekilde geliştirilmiş olması gereklidir (Microsoft, 2021b).

- **Ölçeklenebilirlik**

Mikroservisler bağımsız olarak ölçeklenebilir olmalılardır. Tabii bütün uygulamayı ölçeklemek için daha fazla kaynak gereksinimine ihtiyaç duyulabilir. Bunun ayrıca kubernetes veya docker swarm gibi bir orkestretöre ihtiyaç duyulabilir. Böylelikle sunucudaki kaynakları çok iyi bir şekilde verimli kullanarak çok daha fazla etkili ve performanslı olabilir (Microsoft, 2021b).

- **Veri İzolasyonu**

Veritabanında şema güncellemeleri çok daha kolay olabilir. Çünkü sadece bir mikroservis bu durumdan etkilenir. Monolitik uygulamada ise bu güncelleme biraz daha zor olabilir. Çünkü uygulamanın etkilenen bütün parçalarına dokunmak gerekebilir. Bu şema güncellemelerini daha riskli bir hale getirir (Microsoft, 2021b).

2.1.7. Mikroservislerin Dezavantajları

Mikroservis mimarisinin dağıtık yapısından dolayı karmaşıklık giderek artar. Geliştiriciler mesaj taşıyıcısı (Message Broker) uygulamaları kullanmaya ihtiyaç duyabilirler. Üstelik çalışan bazı servislerde hata meydana gelebilir. Bu nedenle istekler daha yavaş cevaplanabilir veya cevap alınamayabilir. Bu durum monolitik uygulamaya kıyasla çok daha karmaşık bir hal alabilir. Mikroservisler kendi veritabanlarına erişim sağladıkları için veri bütünlüğü ve tutarlığı bağlamında ilave efor sarfedilmesine neden olabilir. Farklı servisler tarafından kullanılan veritabanlarında güncelleme işlemlerimi sırası ile çalıştırmanız gerekebilir. Aynı zamanda mikroservisleri test işlemine tabii tutmakta karmaşık bir hal alabilir. Monolitik uygulamada kullanılan servis metodlarına göre bir sınıf yazılarak bu işlem gerçekleştirilebilir. Ama mikroservisler ile işlem yaparken her servis için ayrı ayrı çalışma yapmak gerekebilir (Richardson , S., 2016).

Mikroservisler ile ilgili bir başka büyük problem ise gerekli olan değişikliklerin birden çok servise uygulanması gerektiğinde ortaya çıkar. Örnek olarak bir servis başka bir servise bağımlı olarak çalışabilir. Bağımlı olduğu serviste başka bir servise bağımlı olarak çalışabilir. Bu durumda her serviste güncelleme

yapıp yayınlamak gerekebilir. Oysa ki bu işlem monolitik uygulamada çok daha basit bir şekilde yapılabilir. Kaynaklar ve bağımlılıklar ortak kullanıldığı için bir kütüphanede yapılan bir değişiklik uygulamanın kalanında kullanılabilir olabilir (Richardson , S., 2016).

Yayınlama işlemleri mikroservis tabanlı uygulamalarda oldukça karmaşık bir hal alabilir. Monolitik uygulamada bu işlem oldukça basittir. Oluşturulmuş paketi sunucuya kopyalayarak işlemi tamamlayabilirsiniz. Ama mikroservislerde ise durum bu kadar kolay değildir. Her servisi ayrı ayrı yayınlamanız gerekebilir. Her servisin birden fazla kopyası sistemde çalışıyor olabilir. Her bir parça ayrı ayrı yayınlanmaya, konfigüre edilmeye, takip edilmeye ihtiyaç duyabilir. Sonuç olarak Mikroservis yayınlama işlemleri daha fazla kontrol gerektirmektedir. Uygulama tabanlı PaaS sistemler ile servis yayınlama işlemi otomatize hale getirilebilir (Richardson , S., 2016).

Microsoft (2021b) mikroservisler ile ilgili aşağıdaki maddelerde dezavantajlarından bahsetmiştir.

- **Karmaşıklık**

Bir mikroservis uygulaması, monolitik bir uygulamaya göre çok daha fazla haraketli parçaya sahiptir. Her servis oldukça basittir, ama bu durum bütün uygulamaya bakıldığında karmaşık bir hal alabilir.

- **Geliştirme ve Test**

Küçük servis geliştirmek, geleneksel yolla geliştirilen monolitik veya katmanlı uygulamalara göre farklı bir yaklaşım gerektirir. Bu durum diğer bağlı servislerin stabil çalışacağını ve güvenilir olduklarını varsaymak ile ilgilidir. Varolan araçlar her zaman servis bağımlılıkları ile birlikte çalışmayabilir. Servisin sahip olduğu bütün kod blokları boyunca yeniden yapılandırma oldukça zorlaşabilir. Bunun yanında test işlemleri de zorlaşarak

- **Yönetimsel Yetersizlik**

Mikrosevis mimarisi ile geliştirme yaparken merkezi olmayan bir yaklaşım için de olmanın çeşitli avantajları olabilir. Çok farklı geliştirme dilleri veya teknolojiler ile geliştirme yapabilirsiniz. Bu bakım yapmayı oldukça zorlaştırabilir. Takımları çok fazla kısıtlamadan çeşitli standartlar getirilebilir. Buna örnek olarak loglama gösterilebilir. Takımlar geliştirme yaparken oluşan hata veya yapılan işlemleri loglamak için bir standart getirilebilir.

- **Ağ Tıkanıklığı veya Gecikmesi**

Küçük parçalar halinde servis geliştirikmek, bir birleriyle çok fazla iletişim kuran servisler anlamına gelmektedir. Bu servis zinciri bağımlılıkları oldukça fazla uzun olabilir. Bu ek gecikme probleme neden olabilir. Asenkron iletişim şekillleri üzerinde çalışma yapmak gerekebilir.

- **Veri Bütünlüğü**

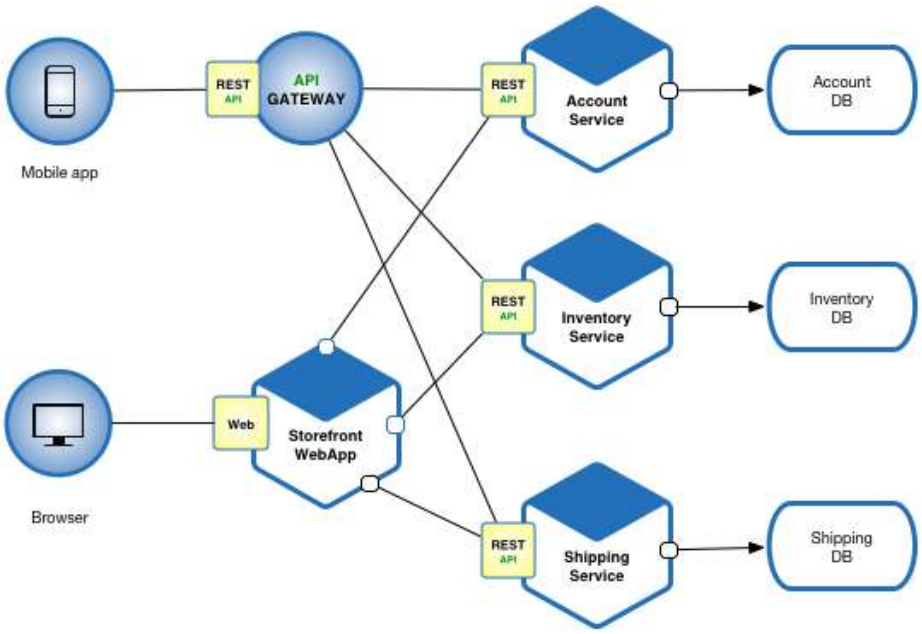
Her mikroservis kendi verisinden sorumludur. Bu durum veri tutarlılığını zorlaştırabilir. Veri tutarlılığını mümkün olduğunca sağlamak için çalışma yapılabilir.

- **Deneyim**

Mikroservisler dağıtık sistemler oldukları için geliştirici takımların deneyimli ve bilgili olması başarıyı artırabilir.

2.1.8. Mikroservislerin Ek Avantajları

Mikroservisler SOAP' ta olduğu gibi xml kullanımı zorunlu değildir. İsteğe göre JSON veya XML kullanabilirler. Mikroservisler farklı dil ve sistemler de geliştirilebilirler ve çok daha kolay birbirleri ile iletişime geçtikleri için etkileşim de zorluk yaşamazlar (Raj ve Ravichandra 2018).



Şekil 25 E-ticaret uygulama mimarisi

Kaynak: Richardson , C. (2021a) ,Pattern: Microservice Architecture , 2021, https://microservices.io/patterns/microservices.html, E.T. 01/12/2021

Gelişen teknoloji ile mobil cihazların kullanımının artması neticesinde ihtiyaçlar değişmiştir. Şekil 4 te görüldüğü üzere farklı sistemler aynı mikroservislere bağlanarak işlemlerini tamamlamaktadırlar. Uygulamanın kullanım yoğunluğuna göre ihtiyaca göre kolaylıkla ölçekleme yapılabilmektedir.

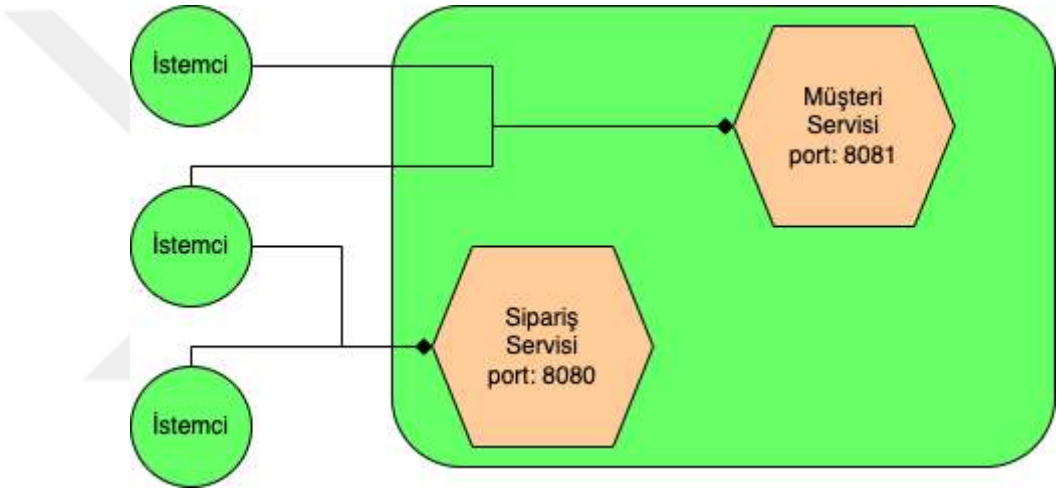
2.1.9. Mikroservis Yayınlama Yöntemleri

Richardson , S. (2016) yayınladığı kitabında aşağıdaki şekillerde mikroservis yayınlama yöntemlerinden bahsetmiştir.

2.1.9.1. Doğrudan Mikroservis ve İstemci İletişimi

Her servisin bir erişim noktası adresi vardır. Bu adres https://servisAdi.api.sirket.adi şelinde olabilir. Bu adrese bir yük dengeleyici yardımıyla servisin birden fazla kopyası çalıştırılarak istemciler yönlendirilir. Fakat bu yöntemin bazı sınırları mevcuttur. Mikroservis tarafından yayınlanan API 'lerin istemci ihtiyaçlarına yeterli cevap veremeyebilir. Bu durum ağ üzerinde istemcilerden gelen çok fazla istek gelebilir, bunun sonucunda internet ortamında yetersiz kalabilir. Bu yaklaşım istemci uygulamalarını daha karmaşık hale getirebilir.

Bu yaklaşımda bir başka problem ise istemciler web dosto protokoller kullanmayabilir. Bir servis ikili RPC kullanabilir, başka bir sevis ise AMQP mesajlaşma protokolünü kullanıyor olabilir. Bu her iki protokol de internet tarayıcıda görüntülenemez veya güvenlik duvarında ilave tanımlama yapılmasına neden olabilir. Uygulamalar, http veya Websocket protokolleri ile internet ortamında bir güvenlik duvarı arkasında erişilebilir olmalıdır. Doğrudan mikroservislere erişimin getirdiği bir başka sorun ise erişilen servislerin birleştirilmesi veya ayrıştırılması gerektirdiğinde karşılaşılır. İstemciler doğrudan servislere eriştiği için bu işlem oldukça zor olabilir. Bu problemler nedeniyle istemciler nadiren doğrudan mikroservislere erişmesi düşünülür. (Richardson , S., 2016) .



Şekil 26 İstemcilerin Mikroservislere Doğrudan Bağlanması

2.1.9.2. Mikroservislerin Docker ile Yayınlanması

Docker ‘in en sağladığı en büyük faydalarından biriside sunucu işletim sisteminden bağımsız olarak uygulamaların bağımlılıklarının birbiri ile çakışmasını önleyebilmesidir. İşletim sisteminden bağımsız çalışan bu uygulamalar, yayınlama aşamasında karşılaşılan problemleri en aza indirmede işlevsellik getirmiştir. Böylelikle CI/CD işlemleri çok daha verimli şekilde yapılabilir olmuştur.

- **Docker nedir?**

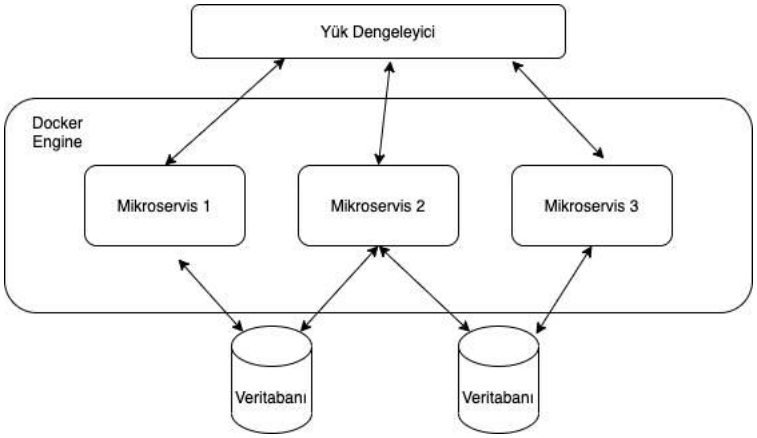
Docker, bir sanallaştırma teknolojisidir. Basit bir sanal makine olarak kabul edilebilir. Geliştiricilere konteyner oluşturmasında ve konteynırlar içinde bulunun uygulamaların yayınlanmasına yardımcı olur (Anderson, 2015).

- **Docker ve Konteynır nedir?**

Konteyner yazılım geliştirme dünyasında hızlı ve etkin bir şekilde Docker kullanarak kolay bir şekilde çalışılabilir hale geliyor. Bununla birlikte çeşitli Linux veya windows tabanlı sistemlerde çalışabilmektedir. Docker, basit olarak uygulamanın bütün bağımlılıklarını içeren evrensel bir paket oluşturur. Docker Engine yardımıyla her hangi bir işletim sisteminde çalışır kılınabilir. Böylelikle geliştiriciyi bağımlılıklarla uğraşma ve "benim bilgisayarımda çalışıyor" gibi durumlardan kurtarır.

Docker uygulama geliştirme, yayınlama ve taşıma işlemlerinin kolaylaştırıldığı bir sanallaştırma uygulamasıdır. Çeşitli araçlar ile bir paket oluşturulur ve geliştiriciye bu paketi kolayca yayınlama imkanı verir (Sarita ve Sebastian, 2017).

Uygulama yük dengeleyici yardımıyla istenilen sayıda çalıştırılabilir böylece SOA'ya göre belirgin bir performans artışı daha elde edilebilir.



Şekil 27 Docker ile Mikroservislerin Yayınlanması

- **Docker ’in 3 önemli bileşeni**

Sarita ve Sebastian (2017) konferansında bu bileşenler aşağıdaki şekilde bahsedilmiştir.

- **Docker İmajı**

Her imajın temel bir imajı vardır ve bu temel üzerine ek bileşenler ilave edilerek oluşturulur. İmajlarda birden fazla katman bulunur. Uygulama güncellemesi yapılacağında bütün imajın yeniden derlenmesi yerine sadece değişiklik yapılan katmanlar derlenir. Docker, yeni bir imaj tanımlamak için Dockerfile dosyasına bakar ve bu dosyaya göre katmanlarını oluşturur. Union File System (UFS) katmanlar arası kontrolün sağlanmasına yardımcı olur.

Docker imajı oluşturmak için Dockerfile oluşturulur.

```
# ilk önce uygulamanın çalışacağı bir temel imaj ile başlanır
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build-env
WORKDIR /app

# proje dökümanları imajın içine kopyalanır ve güncel bağımlılıklar indirilir
COPY *.csproj ./
RUN dotnet restore

# Proje üretim ortamında çalışacak şekilde derlenir
COPY ../engine/examples ./
RUN dotnet publish -c Release -o out
```

İmaj çalıştırıldığında uygulamanın başlandıç noktasının hangi kütüphane olduğu belirtilir.

FROM mcr.microsoft.com/dotnet/aspnet:6.0

WORKDIR /app

COPY --from=build-env /app/out .

ENTRYPOINT ["dotnet", "aspnetapp.dll"]

Docker (2021b)

- **Docker Kayıt Sistemi**

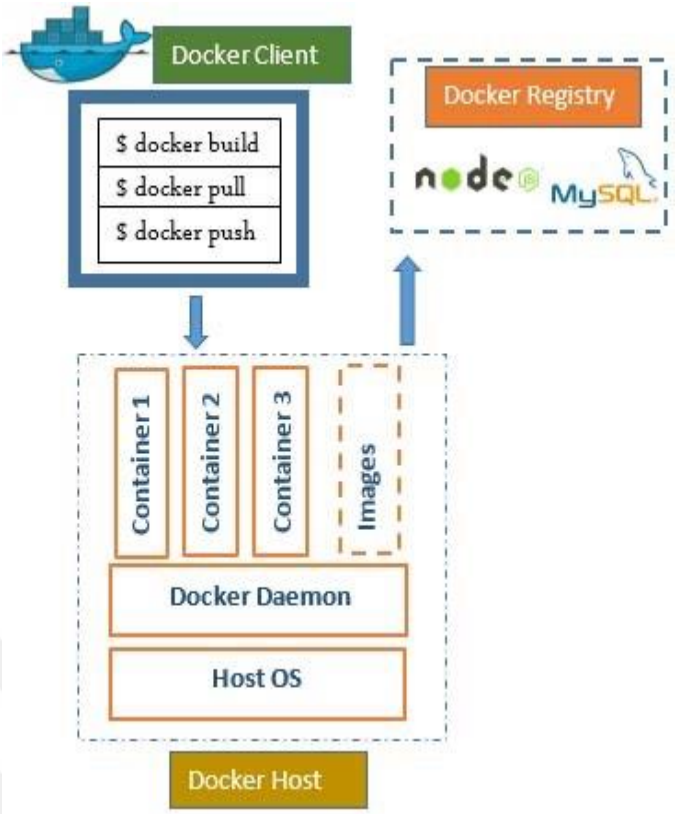
Oluşturulan imajlar kayıt sistemine yüklenir veya sistemden indirilebilir. Docker'in kendine ait Docker Hub adında ortak kayıt sistemi vardır ve bu sisteme imaj yükleme veya sistemden imaj indirme yapabiliriz. Ayrıca bu sistemde imajlarımızı özel olarak saklayabiliriz.

Geliştiriciler aynı zamanda kendi sunucularında Docker kayıt sistemini çalıştırabilirler. Böylece derledikleri imajların kurum dışında değil, kurum içinde saklanmasını sağlayabilirler (Docker, 2021a).

- **Docker Konteynır**

Host işletim sistemi üzerinde sanallaştırılmış, güvenli ve izole edilmiş bir ortam sunar. Bir imaj ile derlenmiş Docker Konteyner'ı kolaylıkla başlatılabilir, durdurulabilir, silinebilir. Konteyner çalıştırıldığında imaj üzerine bir katman daha ekleyerek , uygulama çalıştığında okuma ve yazma işlemlerini yapar.

Docker konteyner oluşturmak için docker komut arayüzü ile run parametresi ile çalıştırılr ve daha önce hazırlanmış olan imajdan bir konteytner oluşturulur. Uygulamanın başlagıç noktası ve kullanacağı port parametre olarak verilerek mikroservis projesi erişime hazır hale getirilir (Docker, 2021b).



Şekil 28 Docker Mimarisi

Kaynak: Sarita ve Sebastian (2017) ,2017 Transform Monolith into Microservices using Docker. 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA), 2017, pp. 1-5.

- **Docker' in Mikroservisler İle Birlikte Kullanımı**

Geliştirilen Mikroservis uygulamasının imajı oluşturulur. Bu imaj, bulut ortamından kolaylıkla indirilebilir ve Docker ortamında bir konteyner aracılığı ile açılır. Böylelikle birbirinden bağımsız olarak servisler çalıştırılır. Ayrıca ihtiyaca göre ölçeklenebileceği gibi her güncelleme de yayınlama işlemi otomatik olarak yapılabilir (Sarita ve Sebastian, 2017).

İşlevselliğini 3 alt başlıkta inceleyebiliriz.

- **Bağımsızlık**

Her Mikroservis belirlenmiş bir konteynır içinde çalışır. Mikroservisin hangi araç veya işlemlerle oluşturulduğundan bağımsız olarak Docker tarafından yayınlanabilir.

- **Otomasyon**

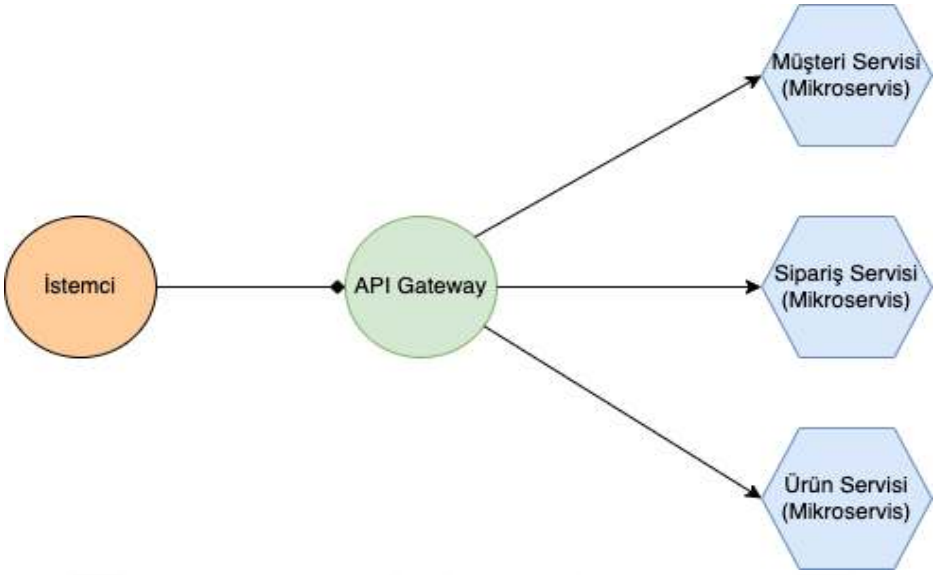
Dockerfile içinde tanımlanan yordamlar ile imaj oluşturması ve konteynır için çalıştırılması mümkün hale gelir. Bu yordamları kullarak birden fazla konteynır oluşturabiliriz.

- **Ölçekleme**

İhtiyaca göre birden fazla Docker konteyner çalıştırarak kolay bir şekilde ölçekleme yapabiliriz.

2.1.9.3. Mikroservislerin Api Gateway ile Yayınlanması

Api Gateway, erişim noktası olan bir sunucudur. Api Gateway iç sunucular arasında bir ara katman oluşturur. Kimlik kontrolü, yük dengeleme, izleme, tamponlama gibi görevleri üstlenebilir.



Şekil 29 Mikroservis API Gateway Kullanımı

Api Gateway, istek yönlendirme, birleştirme, protokol dönüştürme gibi görevlerden sorumludur. Gelen istekleri ilgili servisi yönlendirir. Genellikle iç ortamda birden fazla kopyası çalışan servislere erişerek aradaki iletişimi sağlar. http ve web protokollerinin çevriminde görev alır. Api Gateway bir erişim noktası oluşturur ve istemcilerin verecekleri parametre ile ilgili servisten dönüş almalarını sağlar (Richardson , S., 2016).

2.1.9.4. Api Gateway 'in Avantaj Ve Dezavantajları

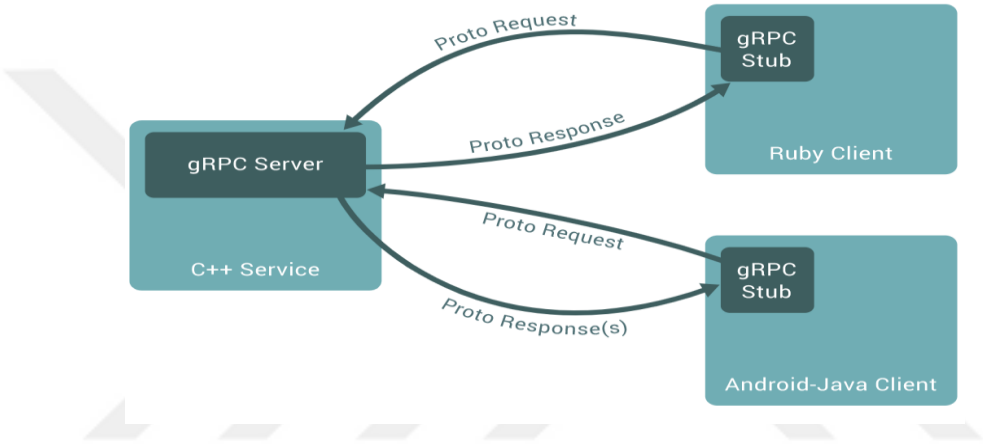
En büyük faydası iç yapı ile istemciler arasında bir katman olarak görev yapar. İstemciler her servis için farklı erişim noktasına bağlanmak yerine Api Gateway aracılığıla iletişim kurarlar. İstemciler için ihtiyaç duyabilecekleri her tür servise erişim imkanı sunarlar. Böylece farklı servisler arasında erişim için uğraşmazlar. Bu aynı zamanda istemci geliştirmesi içinde kolaylık sağlar (Richardson , S., 2016).

Api Gateway 'in en büyük dezavantajı ise istemciler tarafından her zaman erişilebilir olması gerekmektedir. Bu durum aynı zamanda bir dar boğaza neden olabilir. Çünkü istemcilerin hepsi aynı erişim noktasına bağlanmak isteyeceklerdir. Ayrıca geliştiricilerin her yayınlanan servisi Api Gateway 'e tanımlamaları gerekmektedir (Richardson , S., 2016).

2.1.10. gRPC Framework

- **gRPC nedir?**

gRPC, açık kaynak kodlu uzak hizmet çağrısı (RPC) yapabilen her platformda çalışabilen bir framework ‘tür. İstemci ve sunucu uygulama geliştirmeyi destekler ve birbirine bağlı sistemler geliştirilmesini kolaylaştırır. Düşük gecikmeli, ölçeklenebilir dağıtık sistemler geliştirilebilir. Kesin, etkili, dil bağımsız yeni bir protokol geliştirilebilir. İzlenebilen, yetkilendirilebilen, yük dengeleyicisi ile birlikte çalışabilen katmanlı tasarım yapılabilir (Authors, 2021a).



Şekil 30 gRPC Altyapısının Veri İletişim Grafiği

Kaynak: Authors (2021b) Introduction to gRPC, https://grpc.io/docs/what-is-grpc/introduction/ , E.T. 16/12/2021

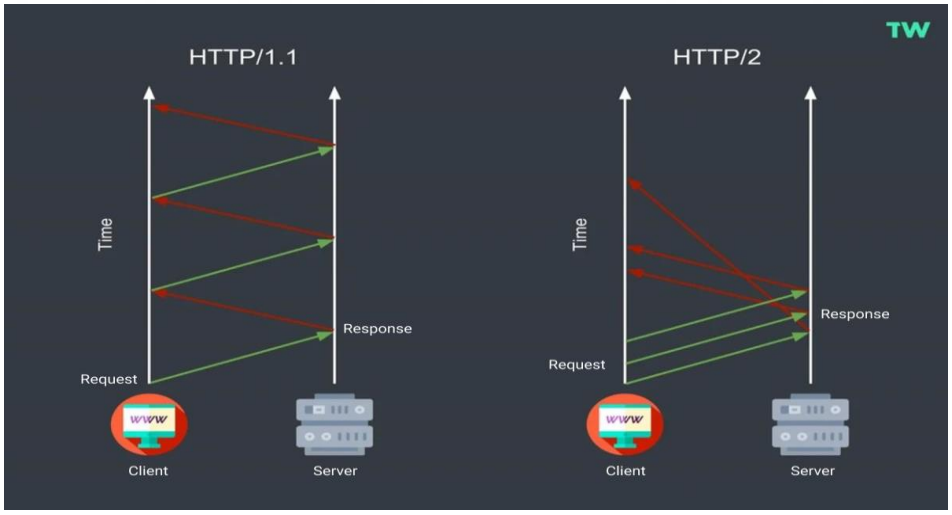
- **gRPC framework’ünün getirdiği yenilikler**

REST mimarisi http/1.1 kullanırken gRPC ise http/2 kullanmaktadır. http/2 , http/1.1 e göre çok daha hızlı çalışmaktadır. gRPC, protocol buffers Google (2021) teknolojisini kullanmaktadır. Bu teknoloji REST mimarisinin kullandığı JSON formatına göre çok daha düşük boyutlara sahip olması nediyle ve enkode veya dekode etmek ile uğraşmadığı için daha performanslı çalışmaktadır. http/1.1 teknolojisinden eş zamanlı istekler desteklenmemektedir. Bu nedenle istekler sırasıyla cevap almaktadır. Bu durumun üstesinden gelinebilmesi için eş zamanlı istekler gönderilsede, yine bu istekler sırasıyla cevaplanmaktadır (Belshe , Peon , ve Thomson 2021).

gRPC, veri transferi yaparken ikili çerçeve protokolünü kullanır ve http 1.1 metin tabanlı olduğu için daha performanslı çalışır. Büyük veri setlerinin asenkron olarak çift yönlü veri akışını destekler. Yapılan isteklerde başlık bilgilerini sıkıştırarak ağ trafiği kullanımını azaltır. JSON serileştirme işleminden çok daha hızlı çalışır (Microsoft, 2021a).

- **Protocol Buffers**

gRPC altyapısı istemci ve servisler arasında iletişim kurarken Protocol Buffers formatını kullanır. Çok daha etkili ve dil bağımsız serileştirme işlemini yapabilmektedir. Geliştiriciler servis kontraktlarını farklı platformlarda oluşturabilirler. Bu kontraktlar .proto uzantılı dökümanlarda oluşturulur. Her servis için metodlar, girdiler ve çıktılar tanımlanabilir. Aynı kontraktlar hem istemci hemde servisler için kullanılabilir. Oluşturulan bu .proto uzantılı dökümanlar protoc uygulaması yardımıyla kullanılan dile çevrilebilir. Veri tipleri kesin bir şekilde tanımlıdır ve istemci ile sunucular arasında paylaşılarak servis operasyonlarını ve veri tiplerini kullanırlar. (Microsoft, 2021a)



Şekil 31HTTP/1.1 ve HTTP/2 istek sırası karşılaştırma

Kaynak: Bayat (2021) A minimalist guide to gRPC , https://itnext.io/a-minimalist-guide-to-grpc-e4d556293422, E.T. 16/12/2021

http/2 de ise istekler eş zamanlı ve asenkron olarak yapılabilmektedir. Bu bir TCP bağlantısında birden fazla istek gönderebileceği anlamına gelmektedir. İstekler bir blok içerisinde bulunur ve bir belirteç ile sunucu ve istemci tarafında işlenir (Belshe , Peon , ve Thomson 2021) .

- **Zayıf Yönleri**
  - **Sınırlı Tarayıcı Desteği**

İnternet tarayıcıları doğrudan gRPC istemcisi gibi çağrı yapamazlar. gRPC büyük oranda http/2 kullandığı için internet tarayıcılar gRPC istemcisi gibi çağrı yapmayı desteklemezler. Bu sorunu çözmek için 2 farklı yaklaşım mevcuttur (Newton-King, 2021).

Bunlar;

gRPC-Web, gRPC takımı tarafından sorunun çözümüne katkısı olması amacıyla geliştirilmiştir. Bu teknoloji ile internet tarayıcıları gRPC' nin sağladığı yüksek performans ve düşük ağ trafiğinin faydalarından yararlanmaktadır. gRPC' nin güçlü özelliklerinden olan çift yönlü veri akışı desteklenmez.

Geliştirilen mikroservis apilerine http metadata ile parametre verilerek cevap olarak gönderilecek veri tipi belirlenebilir. JSON formatında ve .proto formatında servisleri ayrı ayrı yazmadan tercih yapılabilir.

  - **Doğrudan Okunamaz**

Http Api istekleri metin olarak gönderilir, böylelikle geliştiriciler gönderilen veya cevap olarak gelen veriyi rahatlıkla okuyabilirler. gRPC istekleri varsayılan olarak ProtoBuf tipinde enkode edilirek gönderilir. Mesajlar geliştiricilerin okuyamayacağı şekilde ikili biçimde gönderilmektedir. ProtoBuf tipindeki verileri okuyabilmek için ek araçlara ihtiyaç duyulmaktadır.

gRPC ve http Api teknolojileri arasındaki yüksek seviye farklar aşağıdaki gibidir (Newton-King, 2021).

Tablo 3 gRPC ile Mikroservislerin Yüksek Seviye Özelliklerinin Karşılaştırılması

Özellik	gRPC	http Api (JSON)
Kontrakt	Gerekli(.proto)	Seçimli
Protokol	http/2	http/1.1
Veri yükü	Protobuf(küçük, ikili sayı)	JSON
Kuralcı	Katı özellik	Gevşek
Akış	Sunucu, istemci, çok yönlü	Sunucu, istemci
Tarayıcı Desteği	Yok	Var
İstemci Kod oluşturma	Evet	3. Parti araçlar ile

3. PERFORMANS KARŞILAŞTIRMASI

Bu tez kapsamında SOA mimarisinin ve Mikroservis mimarisinin elde edilen performans metrikleri aşağıdaki organizasyonda yapılmıştır. Bu metrikler yardımıyla özgün bir çalışma yapılarak elde edilen performans kıyaslamaları net bir şekilde ortaya konmuştur.

3.1. Ön Çalışma

SOA ile Mikroservisler arasında performans karşılaştırması yaparken kullandığım bilgisayarın sistem gereksinimleri gereği 100.000 gerçek olmayan kişi bilgisi veri çekmesi planlanmıştır. Veritabanı sistemi olarak Microsoft SQL 2020 kullanılmıştır.

3.2. Deneysel Altyapı

Sunucu olarak kullanılan sistem;

İştem Sistemi : macOS Monterey 12.0.1

İşlemci : 2,3 GHz Dört Çekirdekli Intel Core i5

Bellek : 8 GB 2133 MHz LPDDR3

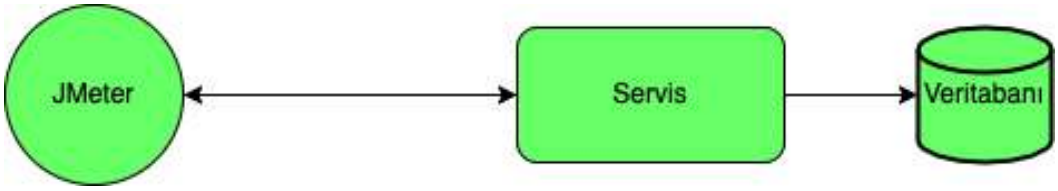
Bant Genişliği : 59.2 GB/s indirme – 6057MB Yükleme

Geliştirme yaptığım bilgisayar macOS Monterey 12.0.1 sistemine sahip olduğu için veritabanı sunucusu DOCKER üzerinde çalıştırılmıştır.

SOA uygulamasını geliştirirken JAVA dili ile Eclipse Uygulaması kullanışmıştır.

Mikroservis uygulaması geliştirirken c# .Net 5.0 ile Vscode uygulaması kullanılmıştır.

Test işlemleri Jmeter uygulaması yapılmıştır. Elde edilen grafikler Jmeter uygulamasının oluşturduğu sonuçlar ile elde edilmiştir.



Şekil 32 JMeter Servis Test Çalışması

Jmeter uygulaması test ortamında Mikroservis'in ve Monolitik servisin istekleri dinlediği bağlantı noktasına istekleri sırası ile göndermektedir. Servisler tarafından yapılan dönüşün ne kadar sürede tamamlandığı ve veri miktarı hakkında bilgi vermektedir.

Monolitik ve Mikroservis mimarisi ile geliştirilen uygulamalar Multi-Thread çalışmaya uygun şekilde geliştirilmişlerdir. Ancak veri çekme performansındaki listelenmiş veriler tek bir kullanıcı tarafından ardışık olarak yapılmıştır.

Servis ilk çalışmaya başlaması için gereken çağırılma süresi servisin büyüklüğüne ve sunucunun donanımsal performansına göre değişiklik göstermektedir. Bu nedenle bu sürenin ölçülebileceği net bir metrik yoktur.

3.3. SOA Uygulamasında Veri Çekme Performansı

3.3.1. Liste Olarak Veri Çekme Performansı

Tablo 4 SOA Veri Çekme Süreleri

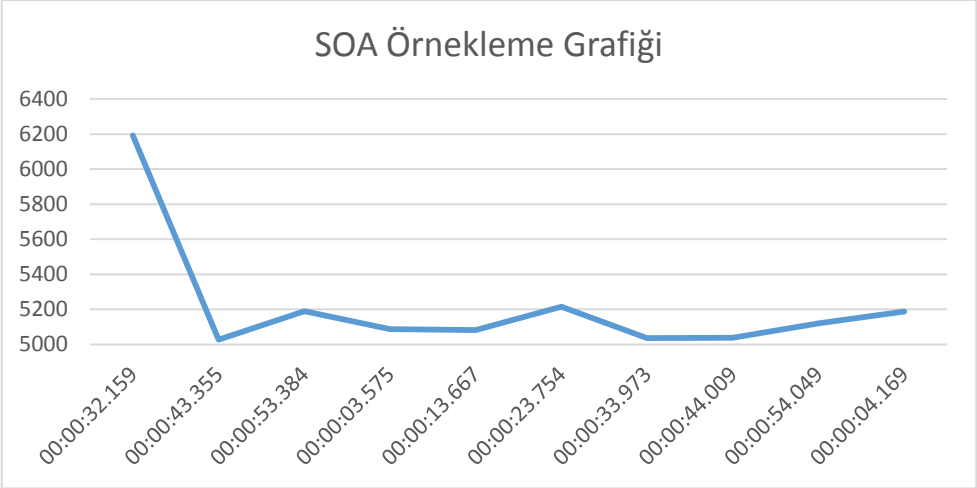
Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
13:09:32.159	HTTP Request (SOAP)	6193	90441811
13:09:43.355	HTTP Request (SOAP)	5027	90441811
13:09:53.384	HTTP Request (SOAP)	5190	90441811
13:10:03.575	HTTP Request (SOAP)	5086	90441811
13:10:13.667	HTTP Request (SOAP)	5082	90441811
13:10:23.754	HTTP Request (SOAP)	5215	90441811
13:10:33.973	HTTP Request (SOAP)	5036	90441811
13:10:44.009	HTTP Request (SOAP)	5038	90441811
13:10:54.049	HTTP Request (SOAP)	5120	90441811
13:11:04.169	HTTP Request (SOAP)	5188	90441811

Listedeki verileri incelediğimizde ortalama 90MB büyüklüğünde bir veriyi 4208 ms ile 5067 ms arasında indirildiğini görüyoruz. Yüklenen veriyi incelediğimizde verinin xml formatında geldiğini görürüz. Verinin sahip olduğu alanların her biri aynı tag ismiyle tanımlanmış ve veri tipi öznitelik olarak verilmiş. Sunucu ile istemci arasındaki iletişim bu şekilde tamamlanmaktadır. Tablo da JMeter test uygulamasıyla yaptığımız işlemler mevcuttur. Bu listede her bir isteğin başlama saati ve bu isteğin tamamlanması için gereken örnekleme zamanını görmekteyiz. Her seferinde aynı miktarda veri talep edilmiş ve oluşturmuş olduğum test ortamında sistemin performansında bağlı olarak farklı sürelerde veri talebinin tamamlandığını görmekteyiz.

Şekil 33 Örnek SOA Verisi

Şekilde görülen örnek XML ile cevaplayan SOA servisidir. Cevap olarak gönderilen veri istemci tarafında kullanılan yazılım dilinin sınıflarına çevrilerek kullanılır. Servislerden dönen veri tipleri güncellendiğinde istemci tarafıda güncel WSDL'e ulaşarak güncellemesini yapar.

3.3.2. Örnekleme Zaman Grafiği



Şekil 34 SOA Örnekleme Grafiği

Grafikte servise 10 adet istek gönderildiği gösterilmektedir. Bu isteklerin gönderildiğini gösteren saatler y düzleminde, cevap olarak gönderilen verinin ne kadar sürede gönderildiği ise x düzleminde gösterilmektedir.

3.3.3. Özet İşlem Performansı

Tablo 5 SOA Servisinin Ortalama Cevap Süreleri

İşlem Adı	Örnekleme Adedi	Ortalama (ms)	Min (ms)	Max (ms)
HTTP Request (SOAP)	10	5217	5027	6193

Tabloda gösterilen grafikteki verilen en düşük sürede verilen cevap ve en yüksek sürede verilen cevap verilmiştir. Bunun yanın SOA servisinin ortalama ne kadar sürede cevap dönüşü yaptığı görülmektedir.

3.4. Mikroservis Veri Çekme Performansı

Senaryoda istemci olarak kullandığımız JMeter uygulaması ile Mikroservise doğrudan bağlantı yapılmaktadır. Mikroservis sistem ortamında kestrel sunucusu ile erişilebilir olmaktadır.

Şekil 35 Mikroservis Senaryo Grafiği

3.4.1. Liste Olarak Veri Çekme Performansı

- **İstek Başlık Bilgileri**

  Connection: keep-alive

  Content-Type: application/json

  Host: localhost:8081

  User-Agent: Apache-HttpClient/4.5.12 (Java/15.0.1)

- **Cevap Başlık Bilgileri**

  HTTP/1.1 200 OK

  Date: Wed, 29 Dec 2021 17:30:37 GMT

  Content-Type: application/json; charset=utf-8

  Server: Kestrel

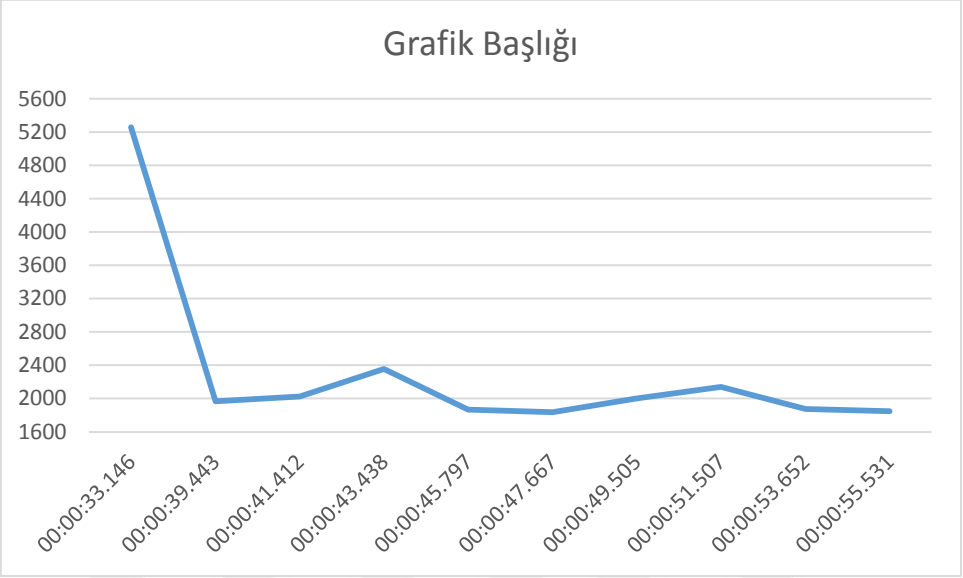
  Transfer-Encoding: chunked

Tablo 6 Mikroservis Veri Çekme Süreleri

Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
13:36:33.146	HTTP Request	5257	37156975
13:36:39.443	HTTP Request	1967	37156975
13:36:41.412	HTTP Request	2025	37156975
13:36:43.438	HTTP Request	2357	37156975
13:36:45.797	HTTP Request	1869	37156975
13:36:47.667	HTTP Request	1837	37156975
13:36:49.505	HTTP Request	2001	37156975
13:36:51.507	HTTP Request	2142	37156975
13:36:53.652	HTTP Request	1877	37156975
13:36:55.531	HTTP Request	1849	37156975

Test işlemini sırasında Mikroservise 10 adet istek gönderilmiştir. Bu isteklerin başlama saatleri, ne kadar sürede cevap aldıkları örnekleme zamanı ile gösterilmiş, gelen veri büyüklüğünün ne kadar olduğuda belirtilmiştir.

3.4.2. Örnekleme Zaman Grafiği



Şekil 36 Mikroservis Örnekleme Grafiği

Şekilde, 10 adet yapılan isteğin ne kadar sürelerde cevaplandığı görüntülenmektedir. Grafikte y düzleminde istedeğin yapıldığı saatler gösterilmekte, x düzleminde cevap olarak dönen verinin ne kadar sürede tamamlandığını göstermektedir.

3.4.3. Özet İşlem Süresi

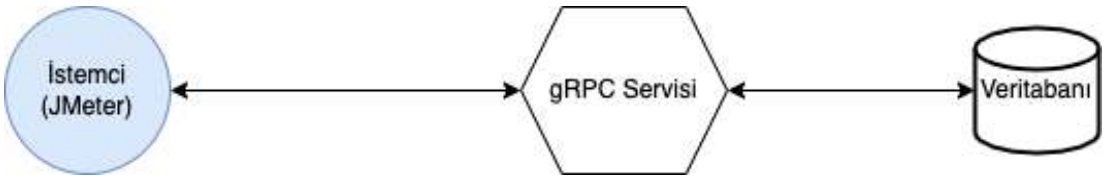
İşlem Adı	Adet	Ortalama (ms)	Min (ms)	Max (ms)
HTTP Request	10	2318	1837	5257

Tablo 7 Mikroservisin Ortalama Cevap Süreleri

Mikroservise 10 adet istek yapıldığı gösterilmektedir. Bu isteklerin en düşük ve en yüksek cevap değerleri listelenmiştir. Böylelikle Mikroservisin ortalama ne kadar sürede cevap döndüğü belirlenmiştir.

3.5. gRPC Uygulamasında Veri Çekme Performansı

Senaryoda istemci olarak kullanılan JMeter uygulaması ile doğrudan gRPC servisine bağlanılmaktadır.



Şekil 37 gRPC İşlem Senaryosu

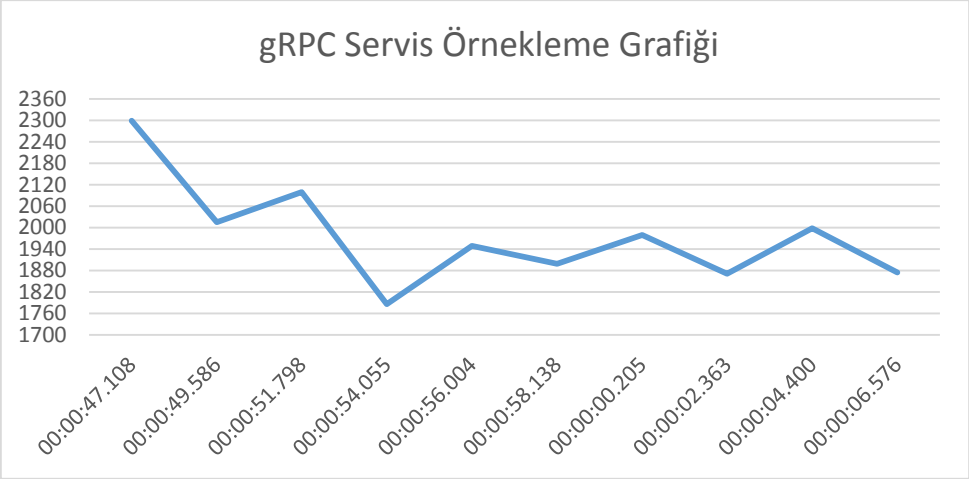
3.5.1. Liste Olarak Veri Çekme Performansı

Tablo 8 gRPC Servisine Yapılan İstekler

Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
13:51:47.108	GRPC Request	2299	8921393
13:51:49.586	GRPC Request	2015	8921393
13:51:51.798	GRPC Request	2099	8921393
13:51:54.055	GRPC Request	1786	8921393
13:51:56.004	GRPC Request	1949	8921393
13:51:58.138	GRPC Request	1899	8921393
13:52:00.205	GRPC Request	1979	8921393
13:52:02.363	GRPC Request	1871	8921393
13:52:04.400	GRPC Request	1998	8921393
13:52:06.576	GRPC Request	1874	8921393

Tabloda gRPC teknolojisi ile 10 farklı istek yapıldığı listelenmiştir. Bu isteklerin hangi saatlerde yapıldığı belirtilmiştir. Yapılan isteklerin ne kadar sürede cevaplandığı ve cevap olarak dönen veri boyutunun ne kadar olduğu gösterilmiştir. Mikroservisten dönen verir ile karşılaştırıldığında çok daha düşük miktarlarda veri transferi gerçekleştiği görülmektedir.

3.5.2. Örnekleme Zaman Grafiği



Şekil 38 gRPC Servisi Örnekleme Grafiği

Şekilde gRPC servise 10 Adet istek gönderilmiştir. Bu isteklerin y düzleminde hangi saatlerde yapıldığı ve x düzleminde ne kadar sürede cevap dönüşü olduğu gösterilmektedir.

3.5.3. Özet İşlem Performansı

Tablo 9 gRPC Ortalama Değerler

İşlem Adı	Adet	Ortalama	Min	Max
GRPC Request	10	1976	1786	2299

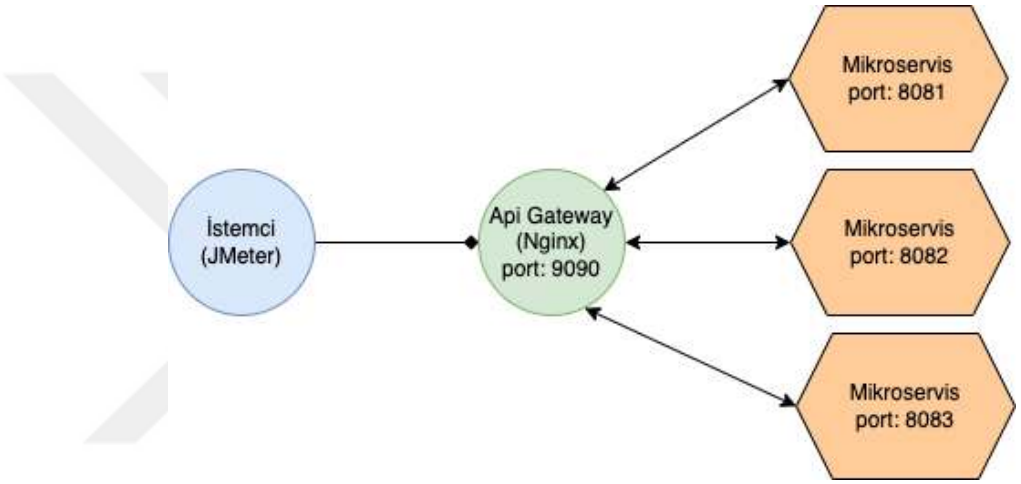
Tabloda, gRPC isteklerinin ortalama ne kadar sürede cavaplandığı görülmektedir. Bununla birlikte servisin en düşük ve en yüksek cevap verme süresi de gösterilmiştir.

Sonuçlar tam bu aşamada Mikroservisler ile karşılaştırılınca tam olarak performans olarak karşılaştırmak mümkün değildir. Bunun nedeni ise gRPC altyapısının çift yönlü yapısının olmasıdır. Test örneğimizde de istek-cevap şeklinde Mikroservis dönüş yapmaktadır. gRPC servisinde de ise benzer işlem yapılmıştır. Fakat bu işlem istek cevap şeklinde yapılmıştır. gRPC nin çift yönlü desteği sayesinde , gRPC servisi tüm listeyi tek seferde göndermek zorunda değildir. Bu

işlemi parça parça yapabilmektedir. Sıkıştırılmış bir veri gönderilbesi neticesinde özellikle anlık veri trafiği gerektiren sistemlerde oldukça işlevseldir.

3.6. Mikroservis Uygulamasında Api Gateway ve Docker Kullanarak Veri Çekme Performansı

Senaryoda yazılımsal yük dengeleyici (nginx) uygulaması kullanıyor. Docker üzerinde çalışan servislere erişim sağlanıyor. Docker üzerinde aynı servisin 3 kopyası çalışıyor. Bu kopyalar yerel makinanın 8081, 8082, 8083 nolu portlarında dinleme yapıyor. Yük dengeleyici ise 9090 nolu porttan istek bekliyor.



Şekil 39 Mikroservis Uygulamasını Api Gateway ve Docker Ortamında Çalıştırma Senaryosu

İstemci olarak kullandığımız JMeter uygulaması yük dengeleyicinin beklediği port olan 9090 nolu porta istek gönderiyor.

3.6.1. Liste Olarak Veri Çekme Performansı

- **İstek Başlık bilgileri**

  Connection: keep-alive
  Content-Type: application/json
  Host: localhost:9090
  User-Agent: Apache-HttpClient/4.5.12 (Java/15.0.1)

- **Cevap Başlık Bilgileri**

  HTTP/1.1 200 OK

  Server: nginx/1.17.4

  Date: Wed, 29 Dec 2021 17:25:50 GMT

  Content-Type: application/json; charset=utf-8

  Transfer-Encoding: chunked

  Connection: keep-alive

Tablo 10 Mikroservise Api Gateway Erişim Performans Tablosu

Başlama Zamanı	İşlem Adı	Örnekleme Zamanı	Veri Boyutu	Gecikme
13:08:31.444	HTTP Request	6673	37177747	4921
13:08:38.119	HTTP Request	5781	37177747	4099
13:08:43.900	HTTP Request	3939	37177747	2484
13:08:47.839	HTTP Request	4308	37177747	2557
13:08:52.148	HTTP Request	4445	37177747	2870
13:08:56.594	HTTP Request	4553	37177747	2995
13:09:01.148	HTTP Request	4161	37177747	2633
13:09:05.309	HTTP Request	4530	37177747	2911
13:09:09.839	HTTP Request	4066	37177747	2498
13:09:13.906	HTTP Request	4272	37177747	2725

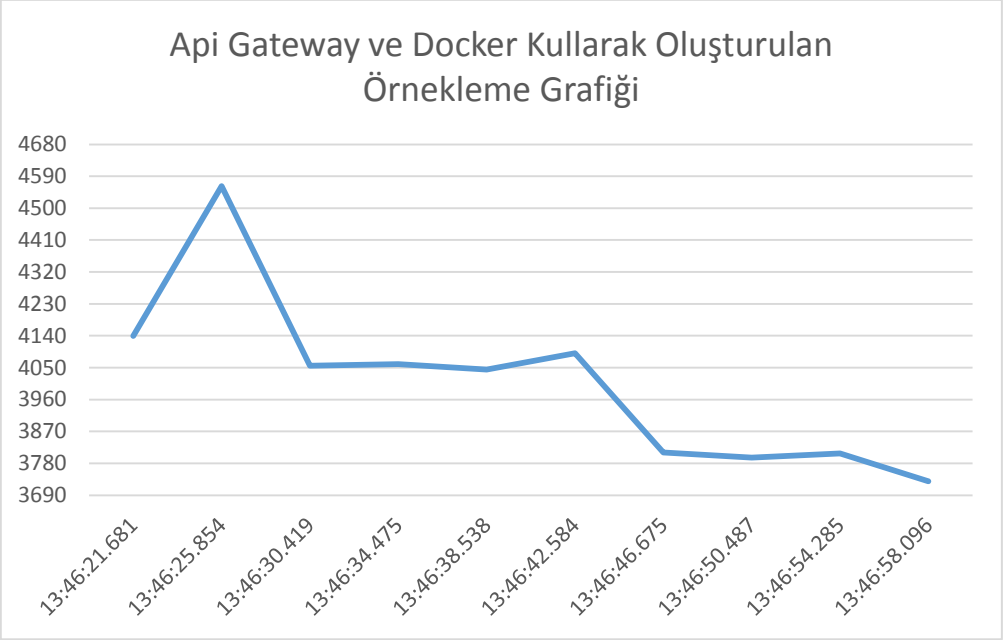
Tabloda, 10 adet istek gönderimi yapıldığı görülmektedir. Başlama zamanı isteklerin yapıldığı saati göstermektedir. Örnekleme zamanı, yapılan isteğin ne kadar sürede tamamlandığını göstermektedir. Veri boyutu yapılan istediğin boyutunun ne kadar olduğunu ifade etmektedir. Gecikme ise istek yapıldıktan sonra dönüş olana kadar geçen süreyi göstermektedir.

İlk istekde tamamlanma süresinin diğer isteklere nazaran daha yüksek olduğu görülmektedir. Bunun nedeni ise servise ilk istek yapıldığında bellekte çalışan bir kopya olmadığı için erişilmek istenen porttaki servisin ayağa kalmaksı beklenmektedir. Sonrasında ise istek sürelerinin giderek düşüşe geçtiği söylenebilir. Burada Mikroservise doğrudan erişime kıyasla daha yüksek değerler elde edildiği söylenebilir. Bunun nedeni olarak test ortamında kullanılan sistem donanımının

yeterli olmaması gösterilebilir. Bir diğer neden olarak istemci hep aynı servise istek gönderdiği için servise erişme isteği ve sayısı arttıkça ağ darboğazına düşüldüğü söylenebilir. Bu durumda donanım ve yazılımsal çalışmaların benzer seviyelerde yapılması gerektiği ortaya çıkabilir.

Ayrıca Mikroservis uygulamasının bir Docker uygulaması içinde bir Konteyner vasıtasıyla çalışması neticesinde ihtiyaç duyduğu bellek ve işlemci miktarı artması neticesinde test ortamındaki sistem donanımının yeterli olmaması nedeniyle performans kaybı yaşadığı görülmüştür.

3.6.2. Örnekleme Zaman Grafiği



Şekil 40 Api Gateway ve Docker kullanarak Oluşturulan Örnekleme Grafiği

Grafikte isteklerin ne kadar sürede tamamlandığı ve sistemde host edilen servislerin hepsi birden ayağa kalktığın cevap süresinin ne kadar kısaldığı görülmektedir.

3.6.3. Özet işlem Performansı

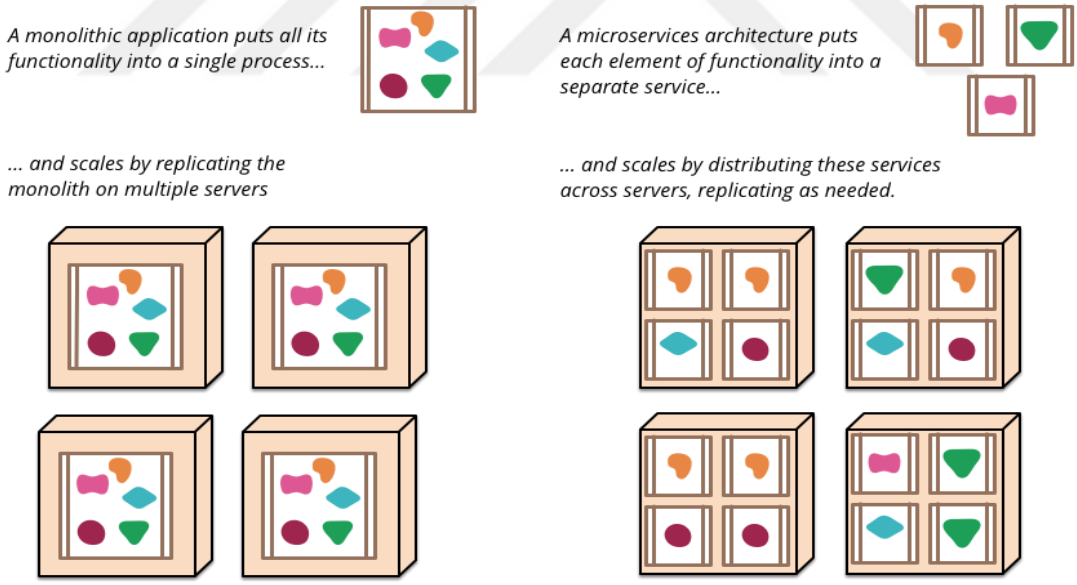
Tablo 11 Ortalama İstek Değeri Tablosu

İşlem Adı	Adet	Ortalama	Min	Max
HTTP Request	10	4009	3729	4562

Tabloda, 10 adet gönderilen isteğin ortalama ne kadar sürede tamamlandığı, en düşük ne kadar sürede ve en yüksek ne kadar sürede tamamlandığı yer almaktadır.

3.7. Son Değerlendirme

Monolitik uygulamalar genelllikle tek bir birim olarak geliştirilir. Her güncelleme yapılışında bütün uygulama yeniden derlenir ve yayınlanır. Bütün işlemler tek bir süreçte çalıştırılır. Monolitik uygulamaları bir yük dengeleyici ile birlikte yatay olarak ölçeklenebilir. Ancak uygulama büyüdükçe bu işlemler zorlaşmaya başlar (Levis, 2014).



Şekil 41 Monolitik ve Mikroservisler Yapısal Karşılaştırma

Kaynak: Microservices, https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa, E.T. 14/12/2021

Mikroservislerde ise bağımsız olarak yayınlanabilir ve ölçeklenebilir. Ayrıca her servis farklı programlama dili ile geliştirilebilir ve farklı takımlar tarafından yönetilebilir (Levis, 2014).

SOA sonuçlarına bakıldığında;

Test ortamında gerçekleştirilen 10 adet istekten en düşük olanı 5027ms ve en yüksek olanı 6193 ms olduğu görülmektedir. Ortalama olarak 5217 ms de isteklerin tamamlandığı görülmektedir. İstek başına indirilen veri boyutuna bakıldığında 90441811 byte olduğu görülmektedir.

Ancak test ortamında geliştirilmiş olan SOA uygulaması başka modülleri içermediği, özellik olarak küçük bir uygulama olması neticesinde performans kaybının belirgin düzeyde olmadığı gözlemlenebilir. SOA uygulaması büyük ölçekli bir hal aldığında sağladığı performas belirgin bir düzeyde düşeceği düşünülebilir. Bu durumun en belirgin nedeni ise Mikroservis uygulamaları küçük ve rahatlıkla ölçeklebilir olması nedeniyle ek donanım yardımıyla farklı sunucularda çalıştırılabilir. Bunun sonucunda Mikroservis çalıştığı sunucunun işlemci ve bellek gücünü kullarak çok daha performanslı çalışabilir.

Mikroservis sonuçlarına bakıldığında;

Test ortamında gerçekleştirilen 10 adet istekten en düşük olanı 1837 ms ve en yüksek olanı 5257 ms olduğu görülmektedir. Ortalama olarak 2318 ms de isteklerin tamamlandığı görülmektedir. İstek başına indirilen veri boyutuna bakıldığında 37156975 byte olduğu görülmektedir.

Mikroservise doğrundan bağlantı yapıldığı için oldukça performanslı sonuçlar elde edildiği görülmüştür. Bunun en büyük etmenlerinden biri ise SOA uygulamasında olduğu gibi XML kullanmak yerine JSON veri tipi kullanması olarak söylenebilir. XML yapısı itibariyle açma ve kapama başlıkları ihtiva ettiği için karakter dizesi bir veriye dönüş türüldüğüne oldukça büyük bir veri miktarı elde edilmektedir. Oysa JSON veri tipi çok daha küçük miktarladadır ve anahtar, değer şeklinde tutulduğu için çok daha düşük veri miktarları elde edilmektedir.

- **Örnek bir JSON tipinde veri aşağıdaki gibidir.**

```
{"isciler":[
  { "adi":"a", "soyadi":"d" },
  { "adi":"b", "soyadi":"e" },
  { "adi":"c", "soyadi":"f" }
]}
```

Şekil 42 JSON Veri Örneği

Şekildeki gibi veriler bir dizi halinde tutulabilir ve veri tipi hakkında tahminde bulunulabilir.

- **Örnek bir XML tipinde veri aşağıdaki gibidir.**

```
<isciler>
  <isci>
    <adi>a</adi> <soyadi>d</soyadi>
  </ isci >
  < isci >
    < adi >b</ adi > < soyadi >e</ soyadi >
  </ isci >
  < isci >
    < adi >c</ adi > < soyadi >f</ soyadi >
  </ isci >
</isciler>
```

Şekil 43 Xml Veri Örneği

Xml ‘de ise veriler biz dizi halinde tutulamaz ve varolan etiketin alt etiketi halinde tutulur. Veri tipinin karakter veya sayı olduğuna bakılarak veri tipi tahmininde bulunabilir. Ama bunun kesin olarak anlaşılabilmesi için etikete öz nitelik ekleyerek belirtilmesi gerekebilir.

Goyal , Singh , ve Ramkumar (2017) araştırmasına göre bu iki veri tipi karşılaştırıldığında;

Tablo 12 XML ve JSON Tiplerinin Karşılaştırması

XML	JSON
Makine çözümlemesi daha zordur	Makine çözümlemesi daha kolaydır
Veri etkiketlerle tanımlandığı için veri boyutu oldukça artar	Anahtar değer çiftleri kullanıldığı için veri boyutu daha küçüktür
Veri alışverişinde orta seviye bir performans sergiler	Yüksek performans suna
XML veri tiplerini ve dizileri desteklemez	JSON veri tiplerini veri dizileri içerir
XML doküman odaklıdır ve verinin yapısal bileşenleri ile ilgilenir	JSON veri odaklıdır ve XML 'in daha hafif bir sürümüdür.

gRPC sonuçlarına bakıldığında;

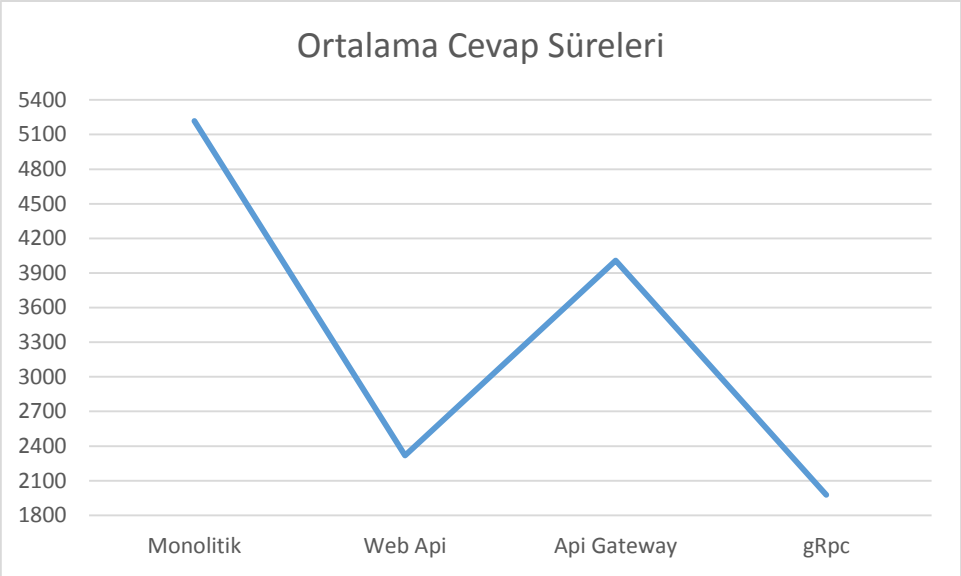
Test ortamında gerçekleştirilen 10 adet istekten en düşük olanı 1836 ms ve en yüksek olanı 3976 ms olduğu görülmektedir. Ortalama olarak 2245 ms de isteklerin tamamlandığı görülmektedir. İstek başına indirilen veri boyutuna bakıldığında 8921393 byte olduğu görülmektedir.

gRPC servis alt yapısı Google (2021) internet sayfasında proto dökümanlarının nasıl oluşturlacağı ile ilgili bilgilendirme yayınlamıştır.

```
message SearchRequest {
  required string query = 1;
  optional int32 page_number = 2;
  optional int32 result_per_page = 3;
}
```

Örnek bir proto dökümanı bu şekilde tanımlandığı gösterilmiştir. Tanımlama işlemlerinden sonra bir ara çeviriçi yardımıyla kullanılan dile çevrilebilmektedir. Desteklediği diller c++, c#, dart, go, java, kotlin ve python 'dur. Örneğin c# kullanıyor isek c# kodu ile servisi çalıştırabilmemiz bir bir kod parçası oluşturur. Geliştiriciler kod parçasını kullanarak servise gelen talepleri kullanılan dilin sınıfları ile alıp gönderebilir. gRPC servisi veri alışverişimi yaparken Protocol Buffers tipine

dönüştürerek yapar. Bu dönüşüm ikili sisteme dönüştürerek gerçekleştirilir. Böylelikle veri boyutu JSON veri tipine göre çok daha aşağılara düşer. Düşen veri boyutu ile performans artışı sağlanabilir. Bu durumun bir dez avantajından bahsedecek olur isek Protocol Buffers formatına dönüştürülen veri JSON gibi insan gözüyle okunamaz.

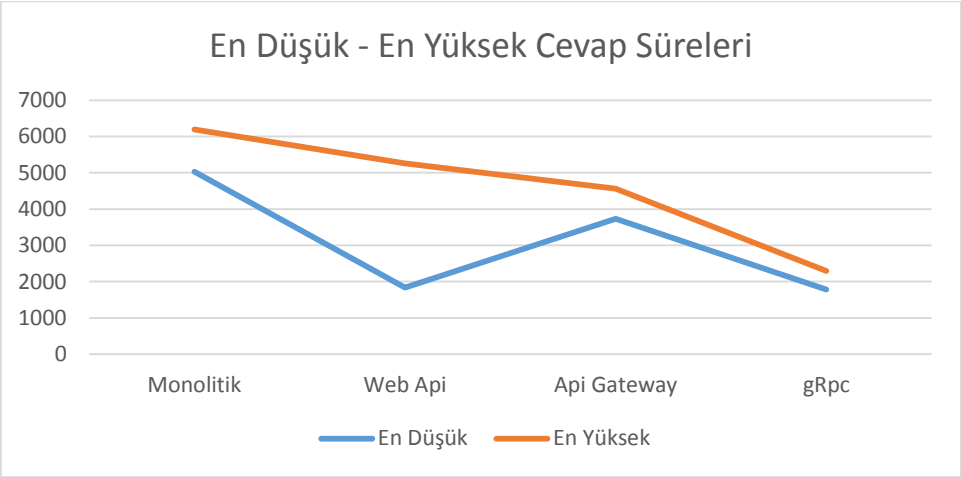


Şekil 44 Ortalama Servis Cevap Süreleri

Grafikte kullanılan servis teknolojilerinin 10 adet çekimde ortalama olarak ne kadar sürelerde cevap döndüğü görülmektedir. SOA ve yük dengeleyici performansları birbirine yakın görünsede istemci sayısının artması ve donanım iyileştirmesi neticesinde belirgin bir fark oluşacaktır.

Mikroservis ve gRPC ortalamalarını ele aldığımızda da benzer sonuçlar elde ettiğimiz görülmektedir. Mikroservis istek-cevap şeklinde çalıştığı için test ortamında da gRPC servisini ölçüm yapabilmek için istek-cevap şeklinde çalıştırılması gerekti. gRPC yapısı itibariyle çok yönlü iletişim sağladığı için cevap olarak gönderilecek veriyi büyük bir liste haline getirip hepsini birden göndermesine ihtiyacı yoktur. Örneğin Mikroservis test örneğinde olduğu gibi 100.000 kayıdı veritabanından alıp topluca gönderim yaparak işlemini tamamladı. Ancak gRPC servisi bu işlemi topluca göndermesine gerek yoktur. İstemciye bu bilgileri küçük paketler halinde gönderebilir. Böylelikle istemci çok büyük verilerde, verilerin gelmesini beklemek yerine istemciye ulaşmış verileri inceleyebilir. Tabii burada ağ

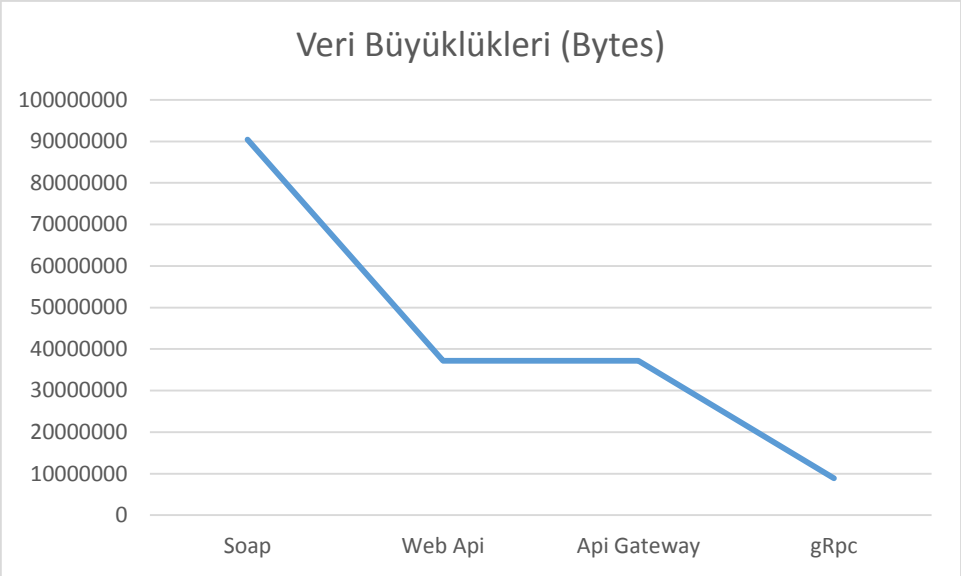
trafiğinin artabileceğinden bahsedilebilir. Çünkü tek bir büyük paket yerine küçük bir çok paket hareket halinde olacaktır.



Şekil 45 En Düşük ve En Cevap Süreleri

Grafikte teknolojilere göre en düşük cevap süreleri ve en yüksek cevap süreleri görülmektedir. Grafiki incelediğimizde SOA ve Yük Dengeleyici servislerinin benzer değerler elde ettiği görülmektedir. Bunun nedeni ise test ortamında kullanılan donanım seviyesinin kaynaklarının sınırlı olması olarak görülebilir. İstemci sayısı arttıkça ve SOA 'ya yeni özellikler ekledikçe kullanılan diğer özellikleride istemciler tarafından kullanılmaya başlandıkça Yük Dengeleyici arkasında çalışan Mikroservislerin çok daha performanslı çalışacağı öngörülebilir. Yük dengeleyicinin başka bir avantajı ise arka planda çalışan Mikroservislerin farklı sunucularda çalıştırılabilir olmasıyla belirgin bir performans kazanımı elde edilebileceği göz önünde bulundurulabilir.

gRPC ve Mikroservis test işlemlerini incelediğimizde benzer sürelerde işlemlerin tamamladığı görülmektedir. Mikroservis yapısı itibariyle istek-cevap şeklinde çalıştığı için bu işlemleri tamamlanması için belirli bir süreye ihtiyacı olduğu söylenebilir. gRPC servisi ise Protocol Buffers tipinde veri ile işlem yaptığı için aslında daha az veri boyutu ile işlem yapmaktadır ve test ortamında da istek-cevap şeklinde ölçüm yapıldığı için benzer sonuçlar elde edilmiştir. Bununla beraber oluşturacakları ağ trafiğini incelediğimizde her iki servisde SOA ya göre çok daha düşük sürelerde işlemlerini tamamladıkları görülmektedir. Bu durumda ağ trafiğinin daha az meşgul edileceği söylenebilir.



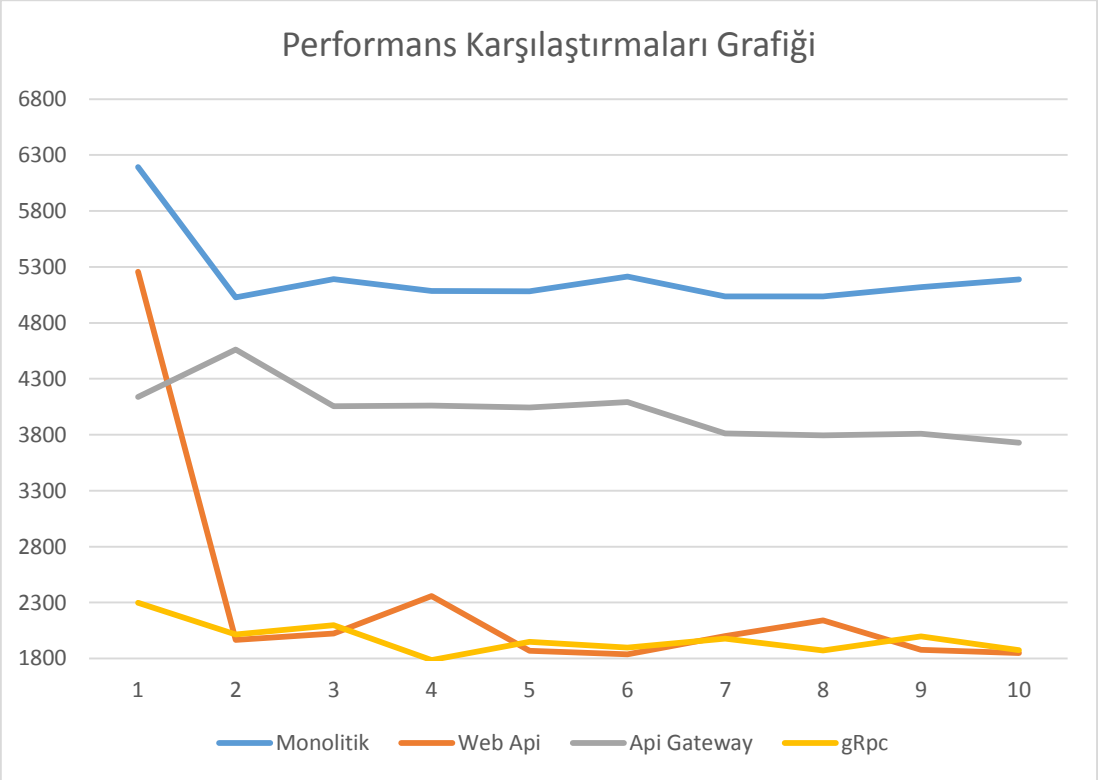
Şekil 46 Veri Büyüklükleri

Grafikte teknolojilere göre yapılan isteğe istinaden cevap olarak gönderilen veri büyüklüğü gösterilmiştir. Grafiği incelediğimizde belirgin farkla gRPC servisinin çok düşük veri miktarları ile işlemini tamamladığı görülmektedir. Bunun nedeni ise Protocol Buffer tipinde veri alışverişi yapabiliyor olmasından kaynaklanmaktadır. Verileri ikili sisteme çevirip göndermesi neticesinde veri boyutunda çok büyük performans kazanımları elde edilmektedir.

Grafikte Mikroservis ve Yük dengeleyici arkasında çalışan Docker Konteyner verilerinin eşit olduğunu görmekteyiz. Aslında her iki testte aynı servis çalışmaktadır. En büyük fark olarak yük dengeleyici testinde istemci servise doğrudan erişim sağlamamaktadır. Tabii burada göz önünde bulundurulması gereken başka bir husus vardır. İstemci sayısı arttıkça yük dengeleyici arka planda bekleyecen 3 adet Mikroservise istekleri eşit oranda yönlendirecektir. Bu durumda istemcilerin bekleme süresi belirgin düzeyde düşme eğilimi gösterecektir. Doğrudan Mikroservise erişim işlemin yine istemci sayısı arttığında Mikroservis gelen istekleri sırası ile cevaplayı deneyecek ve bunun neticesinde gelen sonraki istekler işleme alınması için bir süre beklemek zorunda kalabilecekleri öngörülmektedir.

SOA servisini incelediğimiz veri boyutunun diğer servislere göre çok daha büyük olduğu görülmektedir. Bu sonucun elde edilmesinde en büyük payın XML veri tipi işlem yapılması olarak görülebilir. Çünkü xml birbirini tekrarlayan etiketler

ile veri kapsayarak işlem yapmaktadır. Bu nedenle elde edilen XML dökümanı oldukça büyük miktarlarda veriye ulaşmaktadır. Bu durumun ağ trafiğindeki etkisini göz önüne aldığımızda veri boyutunun fazla olması neticesinde ağ trafiğine ciddi yük getireceği ön görülebilir. SOA servisine yeni özellikler eklendiğini varsayarsak ve bu özelliklere de farklı istemcilerin erişmeye çalışabileceğini düşünürsek belirgin bir performans kaybı yaşanacağı düşünülebilir.



Şekil 47 Performans Karşılaştırmaları Grafiği

Grafikte Monolitik, Web Api, Api Gateway ve gRpc servislerinin performans karşılaştırmalarını görülmektedir. Monolitik uygulama yüksek dönüş süresi ile işlem yaparken Web Api daha performanslı bir süreç izlemiştir. Api Gateway ,doğrudan web api ye erişim sırasındaki gibi performans göstermediği görülmüştür. gRpc servisi ise performans bakımından en verimli seviyelerde işlem yaptığı görülmektedir.

SONUÇ

Gelişen teknoloji ile dağıtık mimaride de çeşitli ihtiyaçlar neticesinde gelişim gerekliliği ortaya çıkmıştır. Mobil uygulamaların kullanımının artması, Iot kullanımının yaygınlaşması, sosyal medya uygulamalarının yaygınlaşması, sağlık sisteminde teknoloji kullanımının yaygınlaşması ve e-ticaret sistemlerinin giderek daha yaygın kullanılmasıyla daha basit bir şekilde veri alışverişi yapılabilecek, daha az ağ trafiği ile daha fazla veri aktarımını mümkün kılan teknolojilere ihtiyaç gerekli hale geldi.

90 'lı yıllarda SOA mimarisi kullanımı revaçta olan dağıtık sistemlerde geliştiren kurumsal uygulamalar için ve farklı sistemlerin birbiri ile iletişim kurması için kullanıyordu. Mikroservis ise daha yeni bir teknoloji olarak ortaya çıktı ve içinde bir çok servisi barındıran büyük monolitik uygulamalar yerine küçük ve birbiri ile iletişim kurabilen mikroservisler kullanılmaya başlandı. Bunun en büyük avantajları ise birbiri ile bağımlılığı olmayan, bağımsız olarak yayınlanabilen, çok daha kolay ölçeklenebilen sistemler olmasıdır. Aynı işlemci veya bellek kullanarak işlemleri gerçekletiren sistemler yerine farklı sistemlerde çalışabilen ve gevşek bağlı şekilde çalışan sistemler gelişen teknoloji ile artan veri miktarını yönetebilir hale geldi. Ölçekleme işlemleri bağımsız servisler olması nedeniyle gerçek anlamda gerçekleştirilebilir hale geldi. Eski sistemlerde aynı uygulamanın birden fazla kopyasının çalıştırılması ile bir nevi yatay ölçekleme mümkün olsada sunucu donanımının gücü kadar performans gösterebilmekteydi. Oysa Mikroservis mimarisinde her servis farklı sunucuda bağımsız olarak yayınlanabilir olmasının yanında Docker gibi uygulamalarla ölçekleme işlemleri çok daha verimli hale gelmiştir. Bunun yanında gRPC alt yapısının getirdiği çok yönlü bağlantı ile daha düşük veri transferine olanak sağlaması büyük miktarlardaki veriyi daha küçük boyutlarda aktarılmasına imkan sağlamıştır.

Ancak bu açılardan daha önce yapılmış bir kıyaslama yoktu. Yapısal olarak SOA mimarisi ile Mikroservis mimarisi birbirlerine benzese de veri alışverişi yaparken gösterdikleri performans bu tez kapsamında SOA ve Mikroservis uygulamalarından elde edilen cevap süreleri ve cevap olarak dönülen veri miktarı ile

doğrudan erişim yapma, yük dengeleyici kullanma veya Docker kullanma yöntemleri ile performans değerlendirmeleri verilmiş ve yeni teknoloji servis mimari sistemlerinin daha düşük ağ trafiği ile daha yüksek veri aktarılabileceğini gösterme hedefine ulaşılmıştır.

KAYNAKLAR

Al-Debagy , O. ve Martinek , P. (2018, 21-22 Nov. 2018). *A Comparative Review of Microservices and Monolithic Architectures.* Paper presented at the 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI).

Anderson, C. (2015). Docker [Software engineering]. *IEEE Software, 32*(3), 102-c103. doi:10.1109/MS.2015.62

Authors, g. (2021a). FAQ. Retrieved from https://grpc.io/docs/what-is-grpc/faq/

Authors, g. (2021b). Introduction to gRPC. Retrieved from https://grpc.io/docs/what-is-grpc/introduction/

Bayat , H. B. (2021). A minimalist guide to gRPC. Retrieved from https://itnext.io/a-minimalist-guide-to-grpc-e4d556293422

Bean , J. (2010). Core SOA Principles. In J. Bean (Ed.), *SOA and Web Services Interface Design* (pp. 25-41). Boston: Morgan Kaufmann.

Belshe , M. ve Peon , R. ve Thomson , M. (2021). Hypertext Transfer Protocol Version 2 (HTTP/2)

. Retrieved from https://httpwg.org/specs/rfc7540.html

Darmawan , B. ve Richter , C. ve Lima , G. ve Salazar , M. E. O. ve Poppe , M. ve Veetil , V. E. (2008). *Power Systems and SOA Synergy*.

Docker. (2021a). Deploy a registry server. Retrieved from https://docs.docker.com/registry/deploying/

Docker. (2021b). Dockerize an ASP.NET Core application. Retrieved from https://docs.docker.com/samples/dotnetcore/

Gehlen, G. ve Pham, L. (2005, 6-6 Jan. 2005). *Mobile Web services for peer-to-peer applications.* Paper presented at the Second IEEE Consumer Communications and Networking Conference, 2005. CCNC. 2005.

Google. (2021). Protocol Buffers Language Guide. Retrieved from https://developers.google.com/protocol-buffers/docs/overview

Gos , K. ve Zabierowski , W. (2020, 22-26 April 2020). *The Comparison of Microservice and Monolithic Architecture.* Paper presented at the 2020 IEEE

XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH).

Goyal , G. ve Singh , K. ve Ramkumar , K. R. (2017, 5-6 May 2017). *A detailed analysis of data consistency concepts in data exchange formats (JSON & XML).* Paper presented at the 2017 International Conference on Computing, Communication and Automation (ICCCA).

He , H. (2003). What Is Service-Oriented Architecture. Retrieved from https://www.xml.com/pub/a/ws/2003/09/30/soa.html

Kumari , S. ve Rath , S. K. (2015, 10-13 Aug. 2015). *Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration.* Paper presented at the 2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI).

Levis, J. (2014). Microservices, a definition of this new architectural term. Retrieved from https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa

McGovern, J. ve Tyagi, S. ve Stevens, M. E. ve Mathew, S. (2003). WSDL. In J. McGovern, S. Tyagi, M. E. Stevens, & S. Mathew (Eds.), *Java Web Services Architecture* (pp. 133-176). San Francisco: Morgan Kaufmann.

Microsoft. (2021a). gRPC. Retrieved from https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc

Microsoft. (2021b). Microservices architecture style. Retrieved from https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Murugesan, S. (2007). Understanding Web 2.0. *IT Professional, 9*, 34-41. doi:10.1109/MITP.2007.78

Newton-King, J. (2021). Compare gRPC services with HTTP APIs. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-6.0

Raj , V. ve Ravichandra , S. (2018, 18-19 May 2018). *Microservices: A perfect SOA based solution for Enterprise Applications compared to Web Services.* Paper presented at the 2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT).

Ramanathan , R. ve Korte , T. (2014, 11-13 July 2014). *Software service architecture to access weather data using RESTful web services.* Paper presented at the Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT).

Richardson , C. (2021a). Pattern: Microservice Architecture. Retrieved from https://microservices.io/patterns/microservices.html

Richardson , C. (2021b). Pattern: Monolithic Architecture. Retrieved from https://microservices.io/patterns/monolithic.html

Richardson , S. (2016). *Microservices From Design To Deployment*.

Sarita ve Sebastian, S. (2017, 17-18 Aug. 2017). *Transform Monolith into Microservices using Docker.* Paper presented at the 2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA).

Singh , V. ve Peddoju , S. K. (2017, 5-6 May 2017). *Container-based microservice architecture for cloud applications.* Paper presented at the 2017 International Conference on Computing, Communication and Automation (ICCCA).

Thönes , J. (2015). Microservices. *IEEE Software, 32*(1), 116-116. doi:10.1109/MS.2015.11

Villamizar , M. ve Garcés , O. ve Castro , H. ve Verano , M. ve Salamanca , L. ve Casallas , R. ve Gil , R. (2015, 21-25 Sept. 2015). *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud.* Paper presented at the 2015 10th Computing Colombian Conference (10CCC).

EKLER

Ek-1. Monolitik Servisin Asenkron İstek Gönderme Testi

Aşağıdaki senaryoda 3 adet kullanıcı aynı anda 10 adet istek gönderiyor. İsteklerini birer saniye arayla gönderiyor. Bu test senaryosunda monolitik servis asenkron olarak kullanıcılardan gelen isteklere cevap vermiştir.

Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
22:04:39.200	HTTP Request (SOAP)	7813	90441811
22:04:39.862	HTTP Request (SOAP)	7630	90441811
22:04:39.530	HTTP Request (SOAP)	7962	90441811
22:04:52.022	HTTP Request (SOAP)	7438	90441811
22:04:52.501	HTTP Request (SOAP)	7132	90441811
22:04:52.501	HTTP Request (SOAP)	7673	90441811
22:05:04.465	HTTP Request (SOAP)	7499	90441811
22:05:04.637	HTTP Request (SOAP)	7397	90441811
22:05:05.191	HTTP Request (SOAP)	6980	90441811
22:05:17.176	HTTP Request (SOAP)	7582	90441811
22:05:16.967	HTTP Request (SOAP)	7796	90441811
22:05:17.037	HTTP Request (SOAP)	7728	90441811
22:05:29.769	HTTP Request (SOAP)	6753	90441811
22:05:29.769	HTTP Request (SOAP)	6871	90441811
22:05:29.762	HTTP Request (SOAP)	6878	90441811
22:05:41.530	HTTP Request (SOAP)	6744	90441811
22:05:41.644	HTTP Request (SOAP)	6643	90441811
22:05:41.644	HTTP Request (SOAP)	6697	90441811
22:05:53.278	HTTP Request (SOAP)	6746	90441811
22:05:53.293	HTTP Request (SOAP)	6737	90441811
22:05:53.347	HTTP Request (SOAP)	6683	90441811
22:06:05.031	HTTP Request (SOAP)	8495	90441811
22:06:05.032	HTTP Request (SOAP)	8494	90441811
22:06:05.032	HTTP Request (SOAP)	8495	90441811
22:06:18.533	HTTP Request (SOAP)	6871	90441811
22:06:18.533	HTTP Request (SOAP)	6697	90441811
22:06:18.533	HTTP Request (SOAP)	5069	90441811
22:06:23.689	HTTP Request (SOAP)	6927	90441811
22:06:23.689	HTTP Request (SOAP)	6927	90441811
22:06:28.606	HTTP Request (SOAP)	5283	90441811

Ek-2. Web Api Asenkron İstek Gönderme Testi

Aşağıdaki senaryoda 3 adet kullanıcı aynı anda 10 adet istek gönderiyor. İsteklerini birer saniye arayla gönderiyor. Bu test senaryosunda web api asenkron olarak kullanıcılardan gelen isteklere cevap vermiştir.

Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
22:19:12.346	HTTP Request	6884	37156975
22:19:12.015	HTTP Request	7205	37156975
22:19:12.678	HTTP Request	6551	37156975
22:19:20.180	HTTP Request	3456	37156975
22:19:20.180	HTTP Request	3516	37156975
22:19:20.180	HTTP Request	3791	37156975
22:19:23.642	HTTP Request	4479	37156975
22:19:23.698	HTTP Request	4506	37156975
22:19:23.975	HTTP Request	4356	37156975
22:19:28.122	HTTP Request	3417	37156975
22:19:28.335	HTTP Request	3488	37156975
22:19:28.208	HTTP Request	3793	37156975
22:19:31.540	HTTP Request	3499	37156975
22:19:31.825	HTTP Request	3249	37156975
22:19:32.002	HTTP Request	3299	37156975
22:19:35.040	HTTP Request	3395	37156975
22:19:35.075	HTTP Request	3775	37156975
22:19:35.303	HTTP Request	3607	37156975
22:19:38.851	HTTP Request	3238	37156975
22:19:38.436	HTTP Request	3881	37156975
22:19:38.912	HTTP Request	3564	37156975
22:19:42.090	HTTP Request	3860	37156975
22:19:42.319	HTTP Request	3868	37156975
22:19:42.477	HTTP Request	3807	37156975
22:19:45.952	HTTP Request	3399	37156975
22:19:46.188	HTTP Request	3579	37156975
22:19:46.286	HTTP Request	3581	37156975
22:19:49.353	HTTP Request	3465	37156975
22:19:49.769	HTTP Request	3239	37156975
22:19:49.868	HTTP Request	3193	37156975

Ek-3. gRpc servise Asenkron İstek Gönderme Testi

Aşağıdaki senaryoda 3 adet kullanıcı aynı anda 10 adet istek gönderiyor. İsteklerini birer saniye arayla gönderiyor. Bu test senaryosunda gRpc asenkron olarak kullanıcılardan gelen isteklere cevap vermiştir.

Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
22:34:51.739	GRPC Request	5951	8921393
22:34:51.927	GRPC Request	5687	8921393
22:34:51.739	GRPC Request	5949	8921393
22:34:58.542	GRPC Request	3781	8921393
22:34:58.542	GRPC Request	4239	8921393
22:34:58.542	GRPC Request	4297	8921393
22:35:02.642	GRPC Request	4276	8921393
22:35:03.240	GRPC Request	4703	8921393
22:35:03.202	GRPC Request	5037	8921393
22:35:08.143	GRPC Request	3217	8921393
22:35:07.502	GRPC Request	4296	8921393
22:35:08.674	GRPC Request	4632	8921393
22:35:11.629	GRPC Request	5645	8921393
22:35:12.499	GRPC Request	5463	8921393
22:35:13.499	GRPC Request	5285	8921393
22:35:17.548	GRPC Request	3830	8921393
22:35:18.339	GRPC Request	4148	8921393
22:35:19.199	GRPC Request	4142	8921393
22:35:21.691	GRPC Request	4380	8921393
22:35:22.856	GRPC Request	4583	8921393
22:35:23.677	GRPC Request	4657	8921393
22:35:26.411	GRPC Request	4448	8921393
22:35:27.798	GRPC Request	4574	8921393
22:35:28.714	GRPC Request	4763	8921393
22:35:31.216	GRPC Request	4646	8921393
22:35:32.801	GRPC Request	4759	8921393
22:35:33.801	GRPC Request	4636	8921393
22:35:36.226	GRPC Request	4560	8921393
22:35:37.896	GRPC Request	4669	8921393
22:35:38.894	GRPC Request	3961	8921393

Ek-4. Api Gateway ile Asenkron İstek Testi

Aşağıdaki senaryoda 3 adet kullanıcı aynı anda 10 adet istek gönderiyor. İsteklerini birer saniye arayla gönderiyor. Bu test senaryosunda Api Gateway ile docker üzerinde koşan 3 adet web api servisine asenkron olarak kullanıcılardan gelen isteklere cevap vermiştir.

Başlama Saati	İşlem Adı	Örnekleme Zamanı	Veri Büyüklüğü (Byte)
22:46:50.857	HTTP Request	11635	37156975
22:46:51.191	HTTP Request	11472	37156975
22:46:51.521	HTTP Request	11820	37156975
22:47:02.541	HTTP Request	7843	37156975
22:47:02.665	HTTP Request	8285	37156975
22:47:03.344	HTTP Request	8822	37156975
22:47:10.387	HTTP Request	9388	37156975
22:47:10.952	HTTP Request	9869	37156975
22:47:12.167	HTTP Request	9516	37156975
22:47:19.780	HTTP Request	8266	37156975
22:47:20.823	HTTP Request	8894	37156975
22:47:21.684	HTTP Request	8992	37156975
22:47:28.047	HTTP Request	9549	37156975
22:47:29.718	HTTP Request	9072	37156975
22:47:30.676	HTTP Request	8634	37156975
22:47:37.598	HTTP Request	8662	37156975
22:47:38.792	HTTP Request	8281	37156975
22:47:39.312	HTTP Request	8511	37156975
22:47:47.074	HTTP Request	8150	37156975
22:47:46.263	HTTP Request	9065	37156975
22:47:47.824	HTTP Request	8803	37156975
22:47:56.629	HTTP Request	7012	37156975
22:47:55.330	HTTP Request	9915	37156975
22:47:55.227	HTTP Request	10061	37156975
22:48:03.653	HTTP Request	8734	37156975
22:48:05.246	HTTP Request	8860	37156975
22:48:05.289	HTTP Request	9099	37156975