MİKROSERVİS MİMARİSİNDE DAĞITIK İŞLEMLERİN YÖNETİMİ

Mustafa GÖRDESLİ
201402204

YÜKSEK LİSANS TEZİ

Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı
Danışman: Prof. Dr. Asaf VAROL

İstanbul
T.C. Maltepe Üniversitesi
Lisansüstü Eğitim Enstitüsü
Aralık, 2022

MİKROSERVİS MİMARİSİNDE DAĞITIK İŞLEMLERİN YÖNETİMİ

Mustafa GÖRDESLİ
201402204
ORCID: 0000-0001-9953-419X

YÜKSEK LİSANS TEZİ
Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı
Danışman: Prof. Dr. Asaf VAROL

İstanbul
T.C. Maltepe Üniversitesi
Lisansüstü Eğitim Enstitüsü
Aralık, 2022



JÜRİ VE ENSTİTÜ ONAYI

Bu belge, Yükseköğretim Kurulu tarafından 19.01.2021 tarihli "Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge" ile bildirilen 6698 Sayılı Kişisel Verilerin Korunması Kanunu kapsamında gizlenmiştir.

ETİK İLKE VE KURALLARA UYUM BEYANI

Bu belge, Yükseköğretim Kurulu tarafından 19.01.2021 tarihli "Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge" ile bildirilen 6698 Sayılı Kişisel Verilerin Korunması Kanunu kapsamında gizlenmiştir.

TEŞEKKÜR

Yüksek lisans tez çalışmamda öncelikle bilgi, katkı ve yardımları için değerli danışman hocam Prof. Dr. Asaf Varol'a ve değerli hocam Dr. Öğr. Üyesi Erdal Güvenoğlu'na, her konuda desteklerini esirgemeyen hep yanımda olan annem Ayşe Gördesli ve babam Ali Gördesli'ye, hem manevi desteğiyle, hem de akademik bilgisiyle bana yön gösteren ablam Dr. Öğr. Üyesi Fatma Pınar Gördesli Duatepe'ye ve son olarak kıymetli eşim Pınar Gördesli'ye ve oğlum Kuzey Gördesli'ye teşekkür ederim.

Mustafa GÖRDESLİ

Aralık, 2022

ÖZET

MİKROSERVİS MİMARİSİNDE DAĞITIK İŞLEMLERİN YÖNETİMİ

Mustafa Gördesli
Yüksek Lisans Tezi
Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Tezli Yüksek Lisans Program
Danışman: Prof. Dr. Asaf Varol
Maltepe Üniversitesi Lisansüstü Eğitim Enstitüsü, 2022

Bu çalışmada mikroservis mimarisi ile oluşturulmuş servisler üzerinde gerçekleşen birbirinden bağımlı ya da bağımsız dağıtık işlemlerin yönetim şekilleri ele alınmıştır. Monolitik yapılarda tüm işlemler tek bir sistem tarafından kontrol edildiği için dağıtık işlemlerin kontrolü de bu servis tarafından rahatlıkla yapılabilmektedir. Ancak, Monolitik yapılardan mikroservis mimarisine geçişte en çok karşılaşılan problemlerden biri, dağıtık işlemlerin birbiri ile ilişkisi olmayan mikroservislerde düzenli bir biçimde nasıl işleneceğidir. Bu probleme getirilen çözümler, mikroservislerin kuyruk sistemi ile olay tabanlı bir şekilde birbiri ile iletişimi ya da yönetici bir servis ile mikroservislerin sıralı ve düzenli bir şekilde kontrol edilmesidir. Bu çözümler daha çok mikroservislerin sıralı bir şekilde çalışması ile ilişkilidir. Bu tezde bu probleme çözüm olarak, çıktı kontrol servisi ile dağıtık işlemlerin paralel bir şekilde mikroservislerin çıktıları işlenmiştir. Bunun yanı sıra dağıtık işlemlerin işlenmesi sırasında oluşabilecek bir hata sonrası yapılacak olan geri alma işlemlerinin nasıl yapılabileceği ele alınmıştır.

Anahtar Sözcükler: Bulut Bilişim, Mikroservis, Servis Tabanlı Mimari, Docker, REST Mimari, Saga Pattern, 2PC

ABSTRACT

DISTRIBUTED TRANSACTION MANAGEMENT IN MICROSERVICE ARCHITECTURE

Mustafa Gördesli
Master Thesis
Graduate School
Computer Engineering Programme with Thesis
Thesis Advisor: Prof. Dr. Asaf Varol
Maltepe University Graduate School, 2022

In this study, the management of dependent or independent distributed transactions in microservice architecture are discussed. Since all transactions in monolithic architecture are controlled by a single system, the control of distributed transactions can be easily done by this service. However, one of the most encountered problems in migration from monolithic architecture to microservice architecture is handling distributed transactions. For the solutions, the event-based communication of microservices with a message queue system or the sequential control of microservices with a manager service can be offered. These solutions were mostly related to the sequential operation of microservices. In this thesis, the solution to this problem is to process the outputs of microservices in parallel with the output control service. In addition, it is also discussed how to make rollback operations after an error that may occur during the processing of distributed transactions.

Keywords: Cloud Computing, Microservices, Event Based Architecture, Docker, REST Architecture, Saga Pattern, 2PC

İÇİNDEKİLER

JÜRİ VE ENSTİTÜ ONAYI ii
ETİK İLKE VE KURALLARA UYUM BEYANI iii
TEŞEKKÜR iv
ÖZET v
ABSTRACT vi
İÇİNDEKİLER vii
TABLOLAR LİSTESİ ix
ŞEKİLLER LİSTESİ x
KISALTMALAR xi
1. GİRİŞ 1
1.1 Yazılım Mimarileri 2
1.1.1 Monolitik mimari 2
1.1.2 Mikroservis mimari 4
1.2 Monolitik ve Mikroservis Mimarilerinin Karşılaştırılması 6
1.3 Literatür Araştırması 7
2. MİKROSERVİS MİMARİSİ TEKNOLOJİLERİ 10
2.1 Konteyner Sanallaştırma Mimarisi 10
2.2 Konteyner Orkestrasyonu 13
2.3 Mikroservislerde Dağıtık Önbellek İşlemleri 14
2.4 Servis Çağırma Yöntemleri 16
2.4.1 Direk servis çağırma yöntemi 16
2.4.1.1 REST mimarisi 18

2.4.1.2 Uzaktan yordam çağrısı (RPC) 20

2.4.2 Kuyruk tabanlı servis çağırma yöntemi 20

2.5 Mikroservis Mimarisinde Dağıtık İşlemlerin Yönetimi 24

2.5.1 İki fazlı işleme (2PC) 25

2.5.2 Saga yöntemi 27

2.5.2.1 Koreografi yaklaşımı 28

2.5.2.2 Orkestrasyon yaklaşımı 29

2.6 Geri Alma (Rollback) Senaryoları 31

2.7 Güvenlik 35

2.7.1 Kimlik doğrulama ve yetkilendirme 36

3. MİKROSERVİS MİMARİSİNDE DAĞITIK İŞLEMLERİN YÖNETİMİNE YENİ ÇÖZÜMLER 38

4. BULGULAR VE TARTIŞMA 45

4.1 Bulgular 45

4.2 Tartışma 46

5. SONUÇ ve ÖNERİLER 48

6. KAYNAKÇA 50

ÖZGEÇMİŞ 56

TABLOLAR LİSTESİ

Tablo 1. Yazılım Mimarileri Özellik Karşılaştırması 46

ŞEKİLLER LİSTESİ

Şekil 1. Monolitik Mimari 3
Şekil 2. Mikroservis Mimari 4
Şekil 3. Docker ve Klasik Sanallaştırma Mimarisi 11
Şekil 4. Klasik Önbellek Veri Çekme ve Yazma İşlemi 15
Şekil 5. Yönetici Servis Altında Çalışan Mikroservisler 17
Şekil 6. gRPC Sunucu ve İstemci Çağırma İşlemi 20
Şekil 7. Mesaj Kuyruğu ile Çalışan Mikroservisler 21
Şekil 8. Kafka Üretim Tüketim Mimarisi 22
Şekil 9. RabbitMQ Exchange Kuyruk Yapısı 23
Şekil 10. 2PC Çalışma Mekanizması 26
Şekil 11. Başarılı Sonuç Dönen Saga Koreografi Yaklaşımı 29
Şekil 12. Başarılı Sonuç Dönen Saga Orkestrasyon Yaklaşımı 30
Şekil 13. Basit Veri Tabanı İlişki Diyagramı 32
Şekil 14. Orkestrasyon Yaklaşımında Geri Alma İşlemi Akış Şeması 33
Şekil 15. Koreografi Yaklaşımında Geri Alma İşlemi Akış Şeması 35
Şekil 16. JWT Çalışma Prensibi 37
Şekil 17. Başarılı Çalışan Mikroservisleri Kontrol Eden Çıktı Kontrol Servisi 39
Şekil 18. Geri Alma İşlemi Yapan Çıktı Kontrol Servisi 41
Şekil 19. Tekrar Deneme İşlemi Yapan Çıktı Kontrol Servisi 42
Şekil 20. Önbellek ile Tekrar Deneme İşlemi Yapan Çıktı Kontrol Servisi 43

KISALTMALAR

MM **:** Monolitik Mimari

MSM **:** Mikroservis Mimari

API **:** Uygulama Programlama Arabirimi (Application Programming Interface)

CPU **:** Merkezi İşleme Birimi (Central Processing Unit)

CRUD **:** Oluştur Oku Güncelle Sil (Create Read Update Delete)

YAML **:** Başka Bir İşaretleme Dili (Yet Another Markup Language)

REST **:** Temsili Durum Transferi (Representational State Transfer)

HTTP **:** Hiper Metin Transfer Protokolü (Hyper Text Transfer Protocol)

URI **:** Tekil Kaynak Belirteci (Uniform Resource Identifier)

RPC **:** Uzaktan Yordam Çağrısı (Remote Procedural Call)

AMQP **:**Gelişmiş Mesaj Kuyruk Protokolü (Advanced Message Queuing Protocol)

JSON **:** Javascript Obje Notasyonu (Javascript Object Notation)

JWT **:** JSON Web Jetonu (JSON Web Token)

CI/CD **:**Sürekli Entegrasyon/Sürekli Kurulum (Continuous Integration/Continuous Delivery)

DevOps **:** Geliştirme ve Operasyon (Development and Operation)

1. GİRİŞ

Yazılım mühendisliğinde, teknolojinin gelişmesi ile yapılan işlemlerin, uygulanan mimarilerin çeşitliliği ve kompleksliliği artmıştır. Daha önceleri küçük çaplı, kompleks olmayan yazılım çözümleri bir firmanın müşterilerini memnun etmesinde yeterli olmakta iken, günümüzde dünyanın her bir yanında milyonlarca müşterisi bulunan firmalar için yetersiz kalmaktadır. Geçen zaman ile kompleksliliği artan yazılım çözümlerinde yeni fikirler, yeni mimariler de ortaya çıkmıştır. Dolayısıyla daha önceleri kullanılan, tüm çözümün tek bir yapıda ele alınıp, tek bir yazılım kod bütününde var olan sistemler, globalleşen dünyada milyonlarca kullanıcıya daha hızlı ve daha güvenli bir biçimde hizmet verebilmesi için yerini zamanla yeni mimarilere bırakmaya başlamıştır.

Yazılımın öncü firmaları olan Google, Amazon, Netflix gibi firmalar yeni nesil yazılım mimarilerinin de öncüleri olmuşlardır. Bununla beraber yeni başlayan küçük firmalar da bu yazılımları ve mimarileri benimseyip uygulamışlardır (Cebeci ve Koçak, 2020). Bu firmaların uygulamaları dünya çapında olduğu ve milyonlarca kullanıcıya aynı zamanda hizmet verdiği için uyguladıkları çözümler, birçok açıdan verimli çözümler olarak ele alınabilir. Ölçeklenebilir, bakımı zor olmayan çözümler olarak bu sistemlerin mimarileri örnek gösterilebilir. Bu mimariler çoğunlukla servis tabanlı mimarilerdir. Tüm sistem tek bir servis tarafından değil, birden çok, mümkün olduğunca bağımsız servisler olarak inşa edilmiştir. Birbirinden bağımsız servislerin var olması daha esnek sistemlerin geliştirilmesi ile ekiplerin birbirinden bağımsız olarak kendi servislerini geliştirip çözümlerini sunmasını sağlamıştır (Serrano, Hernantes, ve Gallardo, 2014).

Yazılımlar kompleksleştikçe, zamanla yeni terimler, yeni yönelimler ortaya çıkmıştır. Bulut bilişim, DevOps gibi kavramlar büyük ve dağıtık mimarileri kullanmak ve kontrol etmek için geliştirilmiş yeni nesil kavramlardır. Bununla beraber, servis olarak yazılım (SaaS, Software as a Service) kavramı da gelişmiştir. Bu da bulut sistemde çalışan ve yazılımı direk kullanılabilen servisler olarak düşünülebilir (Cebeci ve Koçak, 2020).

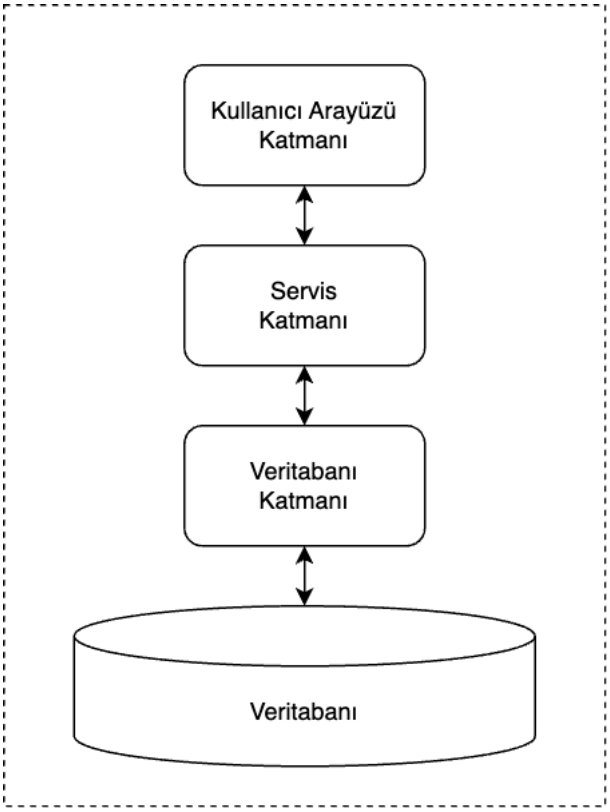
1.1 Yazılım Mimarileri

Bir yazılım projesinde yapılan işlere ve projenin büyüklüğüne göre seçilecek mimari önem taşımaktadır. Küçük çaplı projelerden büyük çaplı çok kullanıcı projelere kadar her bir yazılım projesi belli bir mimari kullanılarak yapılmalıdır. Bunun nedeni sistemin belli bir düzende, doğru ve hızlı bir şekilde çalışmasını sağlamaktır. Günümüzde kurumsal, büyük ölçekli sistemlerde kullanılan yazılım mimarileri, monolitik mimari yani tek bir kod sisteminin yer aldığı tekil bir mimari ve çoklu servislerin yer aldığı servis tabanlı mimarileridir. Bu bölümde, kullanılan yazılım mimarileri incelenecektir.

1.1.1 Monolitik mimari

Monolitik Mimari (MM) tek bir teknolojinin ve kod bütününün kullanıldığı yazılım mimarisidir. Birden fazla kod sistemi olmadığı için, kodun işlenebilir çıktısı bir tanedir. Bu yüzden diğer yazılım mimarilerine göre daha kolay bir şekilde geliştirilebilir. Şekil 1'de görüldüğü gibi MM'de her bir katman kendi içinde ayrı olsa da hepsi beraber bir servisi oluşturmaktadır. Bu katmanlar kendi içlerinde kod üzerinde iletişim halinde olup işlem akışını sağlamaktadır.

Her bir servis kendi içinde ayrı klasörlerde bulunan dosyalar bütünü olarak çalışır. Bir servis başka bir servisi çağıracağı zaman kod içerisinde bunları kontrol eden akış üzerinde yönetilir. Servisler de veri tabanı katmanında kendi veri tabanları ile ilişkili olan kodları çağırarak işlemi tamamlar. Genelde MM'de sadece bir tane veri tabanı kullanılır. Dolayısıyla veri tabanı çok sık işlem alacağı için güçlü bir veri tabanı seçmek gerekebilir.



Şekil 1. Monolitik Mimari

Ancak sistemin zamanla büyümesi, kod sayısının artması gibi nedenlerle, bu tekil mimariyi kontrol etmek zorlaşmaktadır (Kalske, 2018). MM'lerde karşılaşılan problemlerden biri ölçeklenme problemidir. MM'de tüm servisler tek bir çatı altında olduğu için popüler bir servisin trafiği arttığında bu servis tek başına çoğaltılamayacağından, tüm sistemin çoğaltılıp ölçeklendirilmiş olacaktır. Dolayısıyla az trafik alan, çok kullanılmayan servisler de bu popüler servisler ile çoğalmış olacaktır, bu da sistemin gereksiz yere fazla kaynak tüketmesine neden olacaktır (Al-Debagy ve Martinek, 2018).

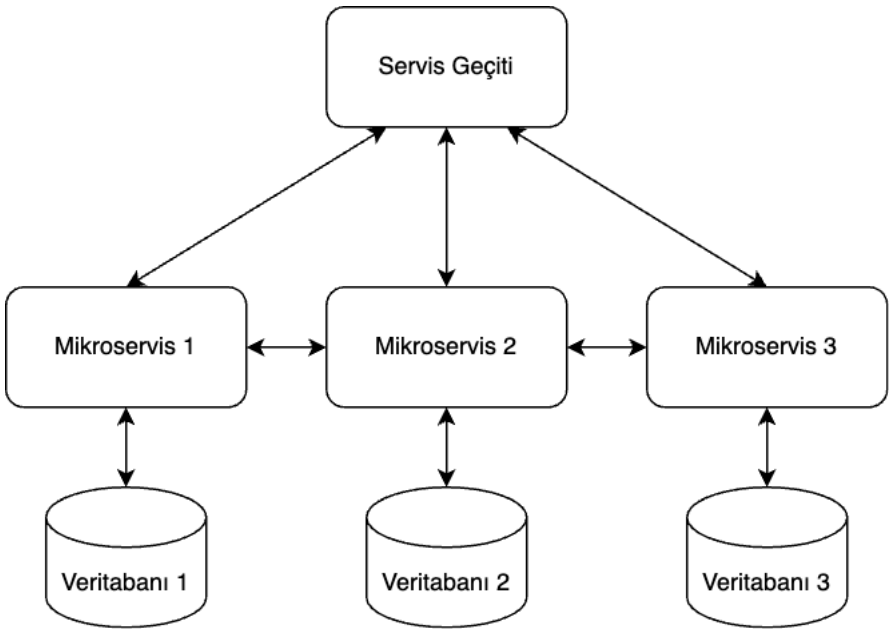
MM'lerde yüzlerce, binlerce servis kod kümesi olarak tek bir çatı altında toplandığı için, farklı takımların bu farklı servislerde iş yapıp daha sonra aynı kod bütününe kodlarını birleştirmeleri, birçok açıdan zorluk teşkil etmektedir (Al-Debagy ve Martinek, 2018). Bu nedenle içerisinde çok servis bulunduran kompleks bir yapı barındıran sistemlerde

monolitik sistem yerine dağıtık mimarili mikroservislerle sistemin yönetilmesi daha uygun olmaktadır.

Tekil kod bütünü olduğu için, yazılım geliştiricilerin yeni bir geliştirme yaptıklarında sistemde hali hazırda var olan daha önce kullanılan servislerin bozulmadığını garanti etmeleri gerekmektedir. Sistem kompleksleştikçe bu garanti verme süreci daha da zorlaşmaktadır ve sisteme yeni bir özellik ekleme işlemleri gittikçe zorlaşmaktadır (Villamizar, ve diğerleri, 2015).

1.1.2 Mikroservis mimari

MM'ye göre daha yeni bir mimari olan mikroservis mimarisi (MSM), daha büyük, kompleks ve rahat ölçeklendirilebilir sistemler geliştirebilmek için oluşturulmuştur. Birbirleri ile farklı yöntemlerle haberleşebilen, kendi içinde bağımsız olan servislerin bir araya gelmesi ile MSM oluşmaktadır (Saransig ve Tapia Leon, 2019). Şekil 2'de bir servisi geçiti kontrolü altında çalışan MSM görseli yer almaktadır.



Şekil 2. Mikroservis Mimari

MSM’de servislerin özellikleri şu şekilde sıralanabilir (Microservies.io İnternet Sitesi, 2022):

Bakımı ve testi kolay yapılabilir.

Bağımsız kurulum yapılabilir.

İş gereksinimine göre parçalanıp yapılandırılabilir.

Bağımsız küçük ekipler tarafından yönetilebilir.

MSM’de her bir servis minyatür bir MM yapısına sahip küçük yapılar olarak düşünülebilir. Yani MM’nin küçük parçalar haline getirilmiş hali denebilir. MM’deki bütün işlem MSM’de küçük parçalara bölündüğü için, her bir sistemin CPU, bellek kullanımı gibi donanımsal parametreler MM’de çalışan bir sisteme göre çok daha düşüktür. MM’de tek bir tane güçlü bir altyapıya sahip bilgisayar kullanılırken, MSM’de daha küçük ölçekli ama daha fazla sayıda bilgisayar kullanılabilir. Bu da maliyet konusunda avantaj sağlayabilir.

Çalışma prensibi MM ile temel olarak aynıdır. Bir işlem MM’de 3 ayrı iş akışından (servis yapısından) geçecek ise, MSM’de de 3 ayrı mikroservis üzerinden işlem yapılıp ana işlem sonuçlanır. Ancak bu mikroservisler birbirinden bağımsız oldukları için, eğer biri hata alır ise, diğerlerini etkilemeyeceği için sistem yoluna devam edebilir.

MSM kullanmanın bir diğer avantajı da bir mikroservisi aynı anda birden fazla çalıştırarak yüksek trafik altında verimli çalışması sağlanabilir. Bu gibi durumlarda yükün dağıtımı yük dağıtıcı (Load Balancer) tarafından kontrol edilir. Yük dağıtıcı bir mikroservise gelen bir isteği ilk karşılar ve o isteği, o sırada çalışan mikroservisler arasından yük olarak en az çalışan mikroservise verir ve sistemin dengesini sağlamış olur.

Bu bağımsız servis yönetimi tüm sistemin hata toleransının yüksek olmasını sağlamaktadır, aynı zamanda servisler bağımsız olduğu için her bir servis kendi içinde diğer servisleri etkilemeden kurulumu yapılabilmektedir. Bu nedenle MM’ye göre MSM’de servislerin kontrolü daha kolay hale gelmektedir (Kratzke, 2015) (Newman, 2015).

1.2 Monolitik ve Mikroservis Mimarilerinin Karşılaştırılması

Monolitik ve mikroservis mimarileri kod bütünlüğü, tekrar kullanılabilirliği, ölçeklenebilmesi gibi konularda karşılaştırılabilmektedir. Bu karşılaştırmalarda her bir mimari diğerine ve kurulacak sisteme göre, avantaj ve dezavantaj sağlayabilmektedir. Sonuçta her iki mimari kullanılarak bir sistem kurulup yönetilebilir, fakat karşılaşılan sorunlar ve bunların çözümleri, hangi mimarinin seçileceğine bağlıdır.

MSM'ler MM'lere göre daha yeni ve popüler bir yazılım mimarisi olarak sunulmaktadır. Genel olarak sistemin yapacağı iş küçük ve ölçeklendirme gibi problemler öngörülmüyor ise MM tercih edilebilir (Cebeci ve Koçak, 2020). Ancak daha büyük sistemlerde, zaman geçtikçe kullanıcı sayısı ve sistemdeki veri miktarı arttıkça sistemi ayakta tutmak, ölçeklendirmek zorlaşabilmektedir. Bu gibi durumlarda ise sistemi olabildiğince küçük iş yapacak parçalara bölüp MSM kullanarak tüm sistemi çalışır hale getirmek daha uygun olabilmektedir (Tapia, ve diğerleri, 2020).

MSM'nin en büyük avantajlarından biri, bağımsız çalışan servislerin bir araya gelip bir bütünlüğü sağlamasıdır. Örneğin bir e-ticaret uygulamasında stok işlemleri, ödeme işlemleri ve kullanıcı işlemlerini ayrı ayrı bir servis olarak ele alıp birbirinden bağımsız şekilde çalışmasını sağlayıp, birbirleri ile haberleşen servisler haline getirmek, bu tüm sistemi MSM'ye geçirmek demektir. Bu sayede hata toleransı yüksek ve yüksek erişime sahip bir yapı haline gelmektedir. Aynı şekilde, küçük bağımsız servislere bölünmüş olan MSM sistemlerde, geliştirme yapmak, test etmek ve sistemin devamlılığını sağlamak MM'ye göre daha rahat olabilmektedir. Her bir ekip kendi servisinin kontrolünü ve devamlılığını sağladığı için bağımsız ekipler rahatça çalışıp genel sistemin de sağlıklı bir şekilde çalışmasını sağlamış olurlar. MM'de ölçeklendirilebilme büyük sorunlardan bir tanesidir. Sistem içinde çalışan birçok servisten sadece birini ölçeklendirmek imkansızdır, çünkü tüm servisler tek bir kod çatısı altında çalışmaktadır ve sistem ölçeklendirilecek ise tüm servislerle beraber ölçeklendirilir. Ancak MSM'de aşırı yük altındaki bir servis bağımsız bir şekilde ölçeklendirilebilir. Diğer yandan, MSM'de var olan birden çok servisin birbiri ile iletişimini kontrol etmek, MM'ye göre oldukça zordur. Çünkü her bir servis bağımsız iş yapabilir, ancak bir servis işi başlatmadan önce başka bir servisin bitmesini bekleyebilir. Bu gibi durumlarda REST iletişim ya da mesaj

kuyruğu gibi iletişimler tercih edilip servislerin birbiri ile iletişimi sağlanabilir. Benzer şekilde mikroservislerin haberleşmesi için Google firmasının kullandığı bir RPC (remote procedural call) türü olan gRPC de son zamanlarda tercih edilmektedir (gRPC İnternet Sitesi, 2022a). MM'de tüm akış tek bir sistemde, tek bir kod bütününde olduğu için, yapılacak işlem kolayca uygulanıp kontrol edilebilir. MM'nin bir diğer avantajı ise kurulum konusunda olmaktadır. MSM'de var olan birçok servisin bağımsız olarak tek tek kurulum yapılıp bunları test etmek, MM'de tek bir sistemin kurulup test edilmesinden çok daha zor olabilmektedir (Nielsen, 2015). Bunun yanı sıra, MSM'de konteyner yapılar ile Docker, Kubernetes gibi uygulamalar, kurulum sürecini kolaylaştıran sistemler olarak kullanılmaktadır (Docker İnternet Sitesi, 2022a) (Kubernetes İnternet Sitesi, 2022a).

1.3 Literatür Araştırması

Literatürde mikroservis ve servis tabanlı mimari kullanılarak birçok araştırma yapılmıştır. Bu araştırmalardan bazıları belli bir sektöre yönelik daha kolay ve uyumlu sistemler olup, bazıları ise bu mimarileri kıyaslayan ve birbirlerine göre avantaj ve dezavantaj çıkarmış olan araştırmalardır.

Cebeci ve arkadaşları kuyruk tabanlı MSM tasarımı ve MM ile performans kıyaslaması yaptığı çalışmada, seçilecek mimarinin iş gereksinimlerine uygun olması gerektiğini vurgulamışlardır, MM ve MSM kıyaslaması yapmışlardır. Ayrıca MM'deki servislerin haberleşme türlerinden bahsetmişlerdir. Sonuç olarak MM ve MSM performans olarak çok yakın sonuçlar vermiş olsa da MSM sayesinde MM'ye göre daha kolay ölçeklenebilir ve yüksek erişilebilirliği olan sistemlerin oluşturulabileceğinden bahsetmişlerdir (Cebeci ve Koçak, 2020).

Kalske ve arkadaşları yaptığı çalışmada MM ile yapılmış bir uygulamayı MSM ile yapılmış bir uygulamaya geçişini konu alan bir çalışma yapmışlardır. Bu çalışmada MM ve MSM konularından bahsetmişlerdir. Ayrıca MM'den MSM'ye geçişte yaşanabilecek sorunları ve zorlukları ele almışlardır. Bu zorlukların başında, kullanılan veri tabanı yönetimi, MSM içerisindeki servislerin iletişimi ve sistemin genel testinin yapılması gelmiştir. Bunun yanı sıra, bir örnek proje ile bu geçişin nasıl yapılabileceğini göstermişlerdir (Kalske, 2018).

Al-Debagy ve arkadaşları yayınladıkları makalede MM ve MSM karşılaştırması yapmışlardır. Bu karşılaştırma çoğunlukla performans tabanlı olmuştur. Eşzamanlı servis çağırma testinde MM'nin, MSM'ye göre %6 oranında daha başarılı olduğunu görmüşlerdir. Yük testi altında ise bu iki mimaride önemli bir fark görmemişlerdir. Ayrıca MSM tasarımında Consul ve Eureka gibi servis keşfetme teknolojilerinden bahsetmişlerdir ve bunları kıyaslamışlardır. Consul'un bu kıyaslamada daha iyi sonuç verdiğini görmüşlerdir (Al-Debagy ve Martinek, 2018).

Villamizar ve arkadaşları MSM modellerini bulut bilişim sistemine uygun bir şekilde nasıl geliştirme, test, kurulum ve ölçeklenme yapılacağından bahsetmişlerdir. Ayrıca büyük internet firmaları olan Amazon, Netflix, Linkedin gibi firmaların büyük ölçekli servislerini hangi çözümler ile kullandıklarından bahsetmişledir. Buna ek olarak, MM ve MSM ile bir örnek proje hazırlayıp bunların performanslarını ve bulut bilişim sistemindeki fiyatlandırmalarını kıyaslamışlardır. Sonuç olarak, yüksek yük altında, MM ile yapılmış uygulamanın tepki süresi MSM ile yapılmış uygulamaya göre daha kısa olduğunu görmüşlerdir. Ancak daha az yük altında MSM'nin tepki süresinin daha kısa olduğunu görmüşlerdir. Ayrıca MSM ile oluşturulmuş sistemin bulut bilişimdeki aylık fiyatının, aynı işi yapan MM ile oluşturulmuş sisteme göre daha düşük olduğunu hesaplamışlardır (Villamizar, ve diğerleri, 2015).

Saransig ve arkadaşları MM ve MSM karşılaştırmasını performans metrikleri üzerinde yapmışlardır. MM ve MSM'lerini birer örnek proje ile oluşturmuşlardır. Projelerinde Node.js ve veri tabanı olarak da MySQL ve MongoDB kullanmışlardır. Mikroservisleri konteyner teknolojisi ile sanallaştırıp Docker üzerinde çalışır hale getirmişlerdir. Çalışmalarında sonuç olarak MSM ile oluşturulmuş örnek projenin MM ile oluşturulmuş örnek projeye göre saniyede daha çok istek işlediğini hesaplamışlardır (Saransig ve Tapia Leon, 2019).

Kratzke yayınladığı makalede mikroservislerin konteyner teknolojisi kullanılarak oluşturulan uygulamalarının ağ üzerindeki performansını ve veri şifreleme performansını incelemiştir. MSM özünde basit, kurulumu kolay sistemler oluşturulmak için kullanılsa da, mikroservislerin çoklanması sonrasında ağ üzerinde oluşabilecek zorluklar ve trafik nedeniyle yönetiminin sanılanın aksine zorlaştığını öne sürmüştür (Kratzke, 2015).

Tapia ve arkadaşları bir örnek proje hazırlayarak MM den MSM'ye geçişi ve bunun performansını kıyaslamışlardır. Hazırladıkları projeyi Amazon Web Servisleri üzerinde kurulum yaparak incelemişlerdir. MSM ile oluşturdukları uygulamayı, Docker ile konteyner yapıya dönüştürmüşlerdir. Yaptıkları kıyaslamaları matematiksel modellere dökmüşlerdir ve sonuçlarında, MM ile oluşturulan uygulamanın bellek kullanımını 4.525155 MB ve MSM ile oluşturulan uygulamanın bellek kullanımını ise 2.887646 MB olarak bulmuşlardır. Ağ üzerindeki kıyaslamada ise, MSM'nin ağ transferi değerini 4.505547 Bytes/s, MM'nin ağ transfer değerini ise 3.129852 olarak bulmuşlardır. Bu sonuçlara göre MSM'nin daha az bellek kullanımı ile gelen isteklere daha hızlı cevap verdiği görülmüştür (Tapia, ve diğerleri, 2020).

Aksakallı ve arkadaşları MSM ile oluşturulmuş uygulamaların kurulumlarının nasıl yapıldıklarını ve mikroservislerin birbirleri ile olan iletişimlerini incelemişlerdir. Çalışmalarında, 3 farklı uygulama kurulum yöntemini ve 7 farklı iletişim yaklaşımını incelemişlerdir. Bu uygulamaların hangi bulut bilişim ortamında, hangi yöntem ile daha önceden incelendiklerini bir kıyaslama tablosu üzerinde göstermişlerdir. Aynı zamanda son yıllarda MSM ile ilgili yayınlanmış makale, bildiri, kitap gibi materyalleri inceleyip hangi türlerde (bildiri, makale vb.) yayınların daha sıklıkla yapıldıklarını incelemişlerdir. Sonuç olarak bu materyallerin daha çoğunlukla konferanslarda yayınlanan makaleler oldukları sonucuna varmışlardır (Aksakalli, Çelik, Can, ve Tekinerdoğan, 2021).

Weerasinghe ve arkadaşları bir bulut bilişim sistemi üzerinde servis tabanlı mimariden MSM'ye geçişi ele almışlardır ve bu mimarilerin ölçeklenebilirliği, performansını ve Amazon Web Servisleri üzerindeki aylık ücretlendirmelerini birer örnek proje ile göstermişlerdir ve bunları kıyaslamışlardır. Sonuç olarak, MSM ile oluşturulmuş uygulamanın eş zamanlı çalışan iş parçacığının, servis tabanlı mimariden daha fazla olduğunu görmüşlerdir. Yani MSM'nin daha hızlı çalıştığını belirlemişlerdir. Buna paralel olarak, MSM ile oluşturulan uygulamanın AWS üzerindeki ücretlendirilmesinin servis tabanlı mimariye göre daha az olduğunu hesaplamışlardır (Weerasinghe ve Perera, 2021).

2. MİKROSERVİS MİMARİSİ TEKNOLOJİLERİ

MSM ile yapılmış sistemlerde karşılaşılan en zorlu işlerden biri de mikroservislerin birbirleri ile iletişimi olmaktadır. Dağıtık işlemlerin yönetiminde, dikkat edilmesi gereken nokta veri bütünlüğünü korumaktır. Birden fazla mikroservisin kullanıldığı bir sistemde, tüm işlemin hatasız bir şekilde bitmesi için veri bütünlüğünün korunması gerekmektedir. Atomik işleme protokolü (Atomic Commit Protocol, ACP) bir sistemdeki veri bütünlüğünü sağlamayı amaçlar. İki fazlı işleme (Two Phase Commit, 2PC) ACP'lerin en çok kullanılanı ve diğer ACP'lerin temelini oluşturmaktadır (Al-Houmaily Y. , 2010). Saga yöntemi de aynı şekilde dağıtık işlemlerin kontrolünde sistemin başarılı sonuç vermesi ve herhangi bir hata alması durumunda veri bütünlüğünü korumayı amaçlar (Garcia-Molina ve Salem, 1987).

Burada bahsedilen veri bütünlüğü, bir sistemin altında çalışan birden fazla mikroservisin veri tabanlarındaki verilerin birbiri ile tutarlı olması halidir. Örneğin, bir tatil sitesinden tatil satın alırken, önyüz ekranında sadece tatili satın al işlemi yaparken, sistem arkada sizin için uçak bileti satın alıp ardından otel odası rezervasyonu yapmaktadır. Bunlar birbirleri ile tamamen bağımsız bir işlemdir. Ancak otel odası rezervasyonu başarılı olup, uçak bileti alma işlemi başarısız olduğunda, sizin için bu otel odası rezervasyonu anlamsız olacaktır. Bu nedenle otel odası rezervasyonu iptali yapılmalıdır. Dolayısıyla veri bütünlüğü için, iki işlemin ya başarılı olması ya da ikisinin de başarısız olması gerekmektedir. 2PC ve saga yöntemi, benzer akışlar kullanarak kullanıcıya bu veri tutarlılığını sağlamaktadır.

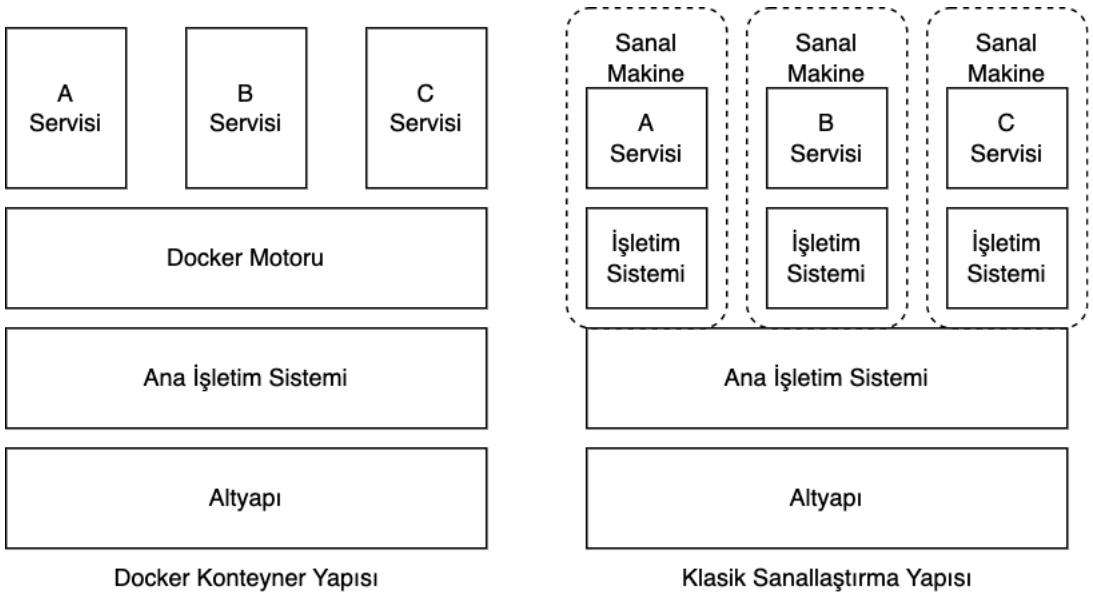
Mikroservis mimarisinde sistemi parçalar halinde bölüp bunları yönetmek için birçok farklı araç ve yöntemler bulunmaktadır. Her çıkan yeni bir teknoloji ya da fikrin ardından bu fikri uygulayan ve yöneten ayrı teknolojiler çıkmaktadır. Bunlardan bazıları konteyner sanallaştırma yöntemleri ve bu konteynerleri yöneten sistemler mikroservis mimarilerinin temellerini oluşturmaktadır.

2.1 Konteyner Sanallaştırma Mimarisi

Sanallaştırma mimarisi birden çok servisi Linux çekirdeği kullanarak birbirinden izole şekilde çalıştırma teknolojisidir. Tek bir sanal bilgisayarda konteyner teknolojisi ile

birbirinden bağımsız servisler oluşturup bunları bağımsız çalıştırmak mümkün olmaktadır (Kocaman, 2018). Günümüzde en yaygın kullanılan sanallaştırma teknolojilerinden biri de Docker'dır. Docker, bir yazılımın çalışması için gerekli tüm çevresel diğer yazılımlarla beraber bir imajının oluşturulması ve ardından bu imajı bir konteyner sistemi üzerinde çalıştırmaya yarayan bir yazılım projesidir. Docker bir imajdan üreyen her bir konteyneri ana makinenin işletim sistemi üzerinde ayağa kaldırır. Klasik sanal makine programlarının aksine her bir konteyner için ayrıca bir sanal makine oluşturmaz (Docker İnternet Sitesi, 2022b).

Şekil 3'te görüldüğü gibi Docker sanallaştırma sistemi ana işletim sistemi üzerine kurulu olan Docker motoru içerisinde A, B ve C servislerini konteyner olarak çalıştırır. Klasik sanallaştırma yapısında ise her bir servis bağımsız ayrı bir sanal makine içerisinde kendine özel tam bir işletim sistemi ile çalışmaktadır. Aynı altyapı klasik sanallaştırma sisteminde birden çok sanal makine çalıştırdığı için daha çok işlem harcar. Bu da Docker konteyner yapısının daha hızlı ve verimli çalışmasını sağlamaktadır.



Şekil 3. Docker ve Klasik Sanallaştırma Mimarisi

Docker projesi ile çalışan konteyner servis oluşturmak için kolaylık sağlayan birçok farklı araçlar vardır. Bunlar, Docker Compose, Docker Hub, Docker Desktop, Docker Machine gibi kavramlardır.

Docker Compose ile aynı bir dosya içerisinde birden fazla Docker imajları ile bir sistemi çalıştırmaya yarayan bir araçtır. Genellikle YAML dosya türünde yazılan Docker Compose dosyaları bilgisayara ayrıca kurulan docker-compose komutu ile çalıştırılır. Örneğin, bir veritabanı ile CRUD işlemi yapan bir servis çalıştırılmak istenirse, Docker Compose YAML dosyası içerisinde iki başlık altında bir servisin gerekli Docker çalıştırma komutu ve kullanılacak veri tabanının çalıştırma komutları yer almaktadır. Böylelikle tek bir hamlede gerekli tüm konteynerler çalıştırılmış olur (Docker İnternet Sitesi, 2022c).

Docker Hub, Docker ile oluşturulan imajların bir çatı altında konteyner kümesi olarak tutulduğu yerdir. Buraya abone olan kişiler, kendi oluşturdukları imajları gönderebilir. Birçok firma da kendi uygulamalarının imajlarını burada tutarlar. Aynı zamanda Docker uygulaması bir konteyner oluşturacağı zaman imajlar bu kümeden çekilir ve ilgili konteyneri çalışır hale getirir (Docker İnternet Sitesi, 2022d). Docker Hub gibi Amazon Web Servisleri, Microsoft Azure gibi bulut bilişim firmalarının da konteynerleri bir küme altında tuttuğu kümeler bulunmaktadır. Bunlar daha çok özel şirketler için kapalı bir küme olarak tutulur. Bir şirkette geliştirilen şirkete özel bir servisi imaj haline getirip şirket içi kullanım için bu kümelerde tutulabilir (Microsoft Azure İnternet Sitesi, 2022) (AWS ECR İnternet Sitesi, 2022).

Docker Desktop, Docker uygulamasının yazılım geliştiricinin kişisel bilgisayarında Docker'ı bir arayüz ile kullanmayı sağlayan bir araçtır. Kullanılan Docker komutlarını bir arayüz yardımı ile fare ile kullanılabilir hale getirir. Docker'ın gerekli tüm özellikleri Docker Desktop uygulaması üzerinden ayarlanabilir. Aynı zamanda Docker Hub ile senkronize olarak oluşturulan imajları imaj kümesine gönderip alma işlemini yapabilir. Docker özünde Linux işletim sistemi ile çalışsa da Docker Desktop ile her türlü işletim sistemine çalışır hale gelebilir (Docker İnternet Sitesi, 2022e).

Docker Machine, hem yerel hem de uzak bilgisayarlarda çalışan birden fazla Docker sisteminin kontrol edilmesini sağlayan bir araçtır. Docker sistemine uzaktan erişim yapabildiği için, kullanılan bilgisayarın özelliğine bakılmaksızın yüksek özellikteki bilgisayarlardaki Docker sistemini kullanmaya yardımcı olmaktadır (Docker İnternet Sitesi, 2022f).

2.2 Konteyner Orkestrasyonu

Büyük ölçekli sistemlerde kullanılan çok sayıda mikroservisler konteyner yapıda tutulup çalıştırılabilir. Yüzlerce ya da binlerce sayıda var olan mikroservisleri kontrol etmek, bir insan tarafından yönetilmeyecek kadar zor olabilir. Bu gibi durumlarda konteyner orkestrasyon kavramı ortaya çıkmaktadır. Örneğin bir e-ticaret sisteminde çalışan arama işlemi yapan bir servis, bir indirim ya da reklam yapıldığında sisteme gelecek çok yüke karşı kendi kendine ölçekleme yapabilmesi gerekmektedir. Konteyner orkestrasyon araçları ile belli bir şart sağlandığında, sistemin belli bir oranda ölçeklenmesi sağlanmaktadır. Örneğin bir servis kümesinde ortalama CPU kullanımı %80'i geçtiğinde %50 oranında büyüme yapılacak şekilde bir ayarlama girilebilir.

Günümüzde kullanılan konteyner orkestrasyon araçlarından biri de Kubernetes'dir. Kubernetes'in dışında bulut bilişim firmalarının da Kubernetes'e alternatif araçları bulunmaktadır. AWS Fargate bunlara örnek olabilir. Kubernetes ölçeklenebilir, açık kaynaklı, konteynerları yönetmeye yarayan aynı zamanda konteyner otomasyonu yapabilen yani servisleri sisteme kurabilen ve yeri geldiğinde geri alma işlemi yapabilen bir araçtır. Kubernetes her zaman çoklu yapıda çalışır. Bir tane ana Kubernetes servisi olup, buna bağlı alt kümeleri ile beraber olası bir hata durumunda, çalışan başka bir alt küme üzerinden işlemine devam edebilir (Kubernetes İnternet Sitesi, 2022b). Kubernetes'in sağladığı diğer avantajlar şunlardır;

Yük dengeleyici olarak kullanılabilir.

Servislerin otomatik kurulmasını ve geri alınmasını sağlar.

Şifreleri ve ayarları tutup bunların yönetimini sağlar.

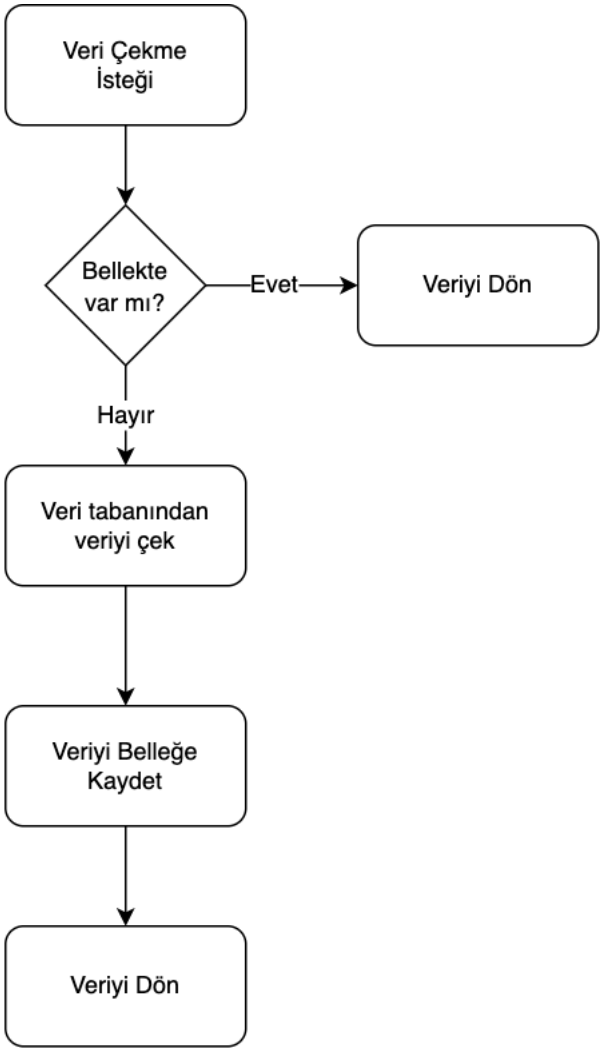
Çoklu yapıda çalıştığı için sistemin erişilebilirliğini artırır.

Kubernetes sıradan bir yazılımcı için çok derin bilgi ve teknik beceri isteyen bir yapı olabilir. Çünkü Kubernetes büyük bir sistemi tamamen ve tek başına yönettiği için ilk başta yapılması gereken ayarlar ve kurulumlar çok önemli ve kompleks olabilmektedir. Buna çözüm olarak bulut bilişim firmaları bir arayüz kullanılarak konteyner orkestrasyonu yapabilen ürünler geliştirmiştir. Bunlardan biri Amazon Web Servislerinin

Fargate ürünüdür. AWS Fargate ile Kubernetes'teki gibi birçok ayarlama yapmadan, sadece arayüz üzerinden olası bir senaryoda sistemin hangi oranda büyüyeceği ya da küçüleceği gibi ayarlamalar yapılabilmektedir (AWS Fargate İnternet Sitesi, 2022).

2.3 Mikroservislerde Dağıtık Önbellek İşlemleri

Önbellek işlemi, bir sistemde sürekli kullanılan bir verinin, o veriyi oluştururken geçen sürenin ve maliyetin azaltılması için bellekte tutulup gerektiğinde o bellekten kullanma işlemine denmektedir (Zhong, Zheng, Liu, Wang, ve Cao, 2020). Örneğin bir veri tabanından bir veri çekilip bu veriyi kullanıcıya dönme işlemi, ilgili servisin tetiklenmesi ardından ilgili servisin veri tabanına istek atması, veri tabanının bu veriyi arayıp bulması ve servise geri dönmesi ardından servisin bu veriyi kullanıcıya dönmesi gibi adımlar içermektedir. Bu işlem veri tabanının veriyi araması ve dönmesi zaman alabilmektedir. Ancak bu işlem çok sık tekrarlanıyor ise yani aynı veri belli bir süre içerisinde sürekli çekiliyor ise, bu veri ilk çekildiği sırada belli bir süre zarfı içerisinde bellekte tutulabilir ve sistem bu veriyi bellekten çekerek veri tabanı işlemini yapmasına gerek kalmaz. Böylelikle sistem daha hızlı bir şekilde çalışmış olacaktır. Şekil 4'te bir servisin önbellek işlemini nasıl yönettiği görülmektedir. Burada bir servise veri tabanından bir çekme isteği geldiğinde öncelikle bellekte var mı yok mu kontrolü yapar. Var ise bellekteki veriyi kullanıcıya döner. Veri yok ise, veri tabanından veriyi çeker, bu veriyi belleğe yazar ve kullanıcıya döner. Bir sonraki aynı istekte artık bellekteki veriyi kullanır.



Şekil 4. Klasik Önbellek Veri Çekme ve Yazma İşlemi

Burada dikkat edilmesi gereken nokta, eğer veri güncellenir ise, güncelleyen sistem bu verinin bellekteki kontrolünü yapmalı ve o veriyi bellekte de güncellemelidir. Aksi taktirde veri tutarsızlığı meydana gelebilir.

Mikroservis mimarinde ise, aynı servisten aynı anda birden çok servis çalıştığı için, her bir servis kendi belleğinde bu veriyi tutup gerektiğinde bu veriyi dönmesi maliyet açısından verimsiz olabilir. Buna çözüm olarak dağıtık önbellek işlemleri yapılmaktadır. Her bir mikroservisin tek bir noktada yönetilen ayrı bir servis ile yönetilen bellek üzerinde işlem yapan bir önbellek servisi ile iletişimde olması ve bunu önbellek işlemi için kullanması, buna bir örnek olarak verilebilir. Bu durumda, sistemde bir veri tabanı işlemi

olduğunda herhangi bir mikroservis bu veriyi veri tabanından çekip, önbellek servisi ile bunu önbelleğe yazar. Daha sonra başka bir mikroservis aynı veriyi çekme işlemi yaptığında, ilk olarak önbellek servisine bu verinin varlığını kontrol eder, eğer var ise veri tabanına gitmez ve direk önbellek servisinden bu veriyi çeker. Böylelikle daha hızlı bir şekilde tepki vermiş olur (Li, Ye, ve Zhou, 2022). Önbellek yönetiminin dağıtık bir mimariye sahip olması, olası bir hatada paralel çalışan önbellek sistemlerin birbiri yerine devreye girmesi olarak özetlenebilir. Böylelikle sistemin erişilebilirliği artmaktadır.

Dağıtık bellek yönetimi yapan teknolojilerden biri Redis'tir. Redis, açık kaynaklı, bellek üzerinde veriyi tutup yöneten, önbellek işlemi yapan aynı zamanda bir veri tabanı olarak da kullanılabilen bir sistemdir (Redis İnternet Sitesi, 2022).

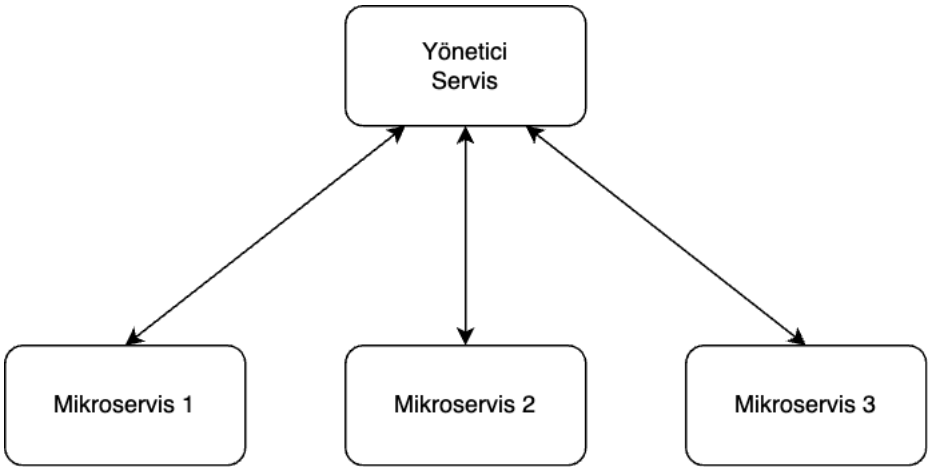
2.4 Servis Çağırma Yöntemleri

Dağıtık işlemlerin yönetiminde öncelikle mikroservislerin birbirlerini nasıl çağırdığına karar vermek önemlidir. Günümüzde firmalar tarafından birçok servis çağırma metodu kullanılmaktadır. Bu metotlar genelde altyapıda iki farklı yöntemi kullanmaktadır. Bunlar, direk servis çağırma yöntemi ve kuyruk tabanlı servis çağırma yöntemi olarak sayılabilir.

2.4.1 Direk servis çağırma yöntemi

Direk servis çağırma yöntemi ile mikroservisler birbirlerini senkron olarak çağırabilmektedirler. Bu yöntemle sistem içerisinde çalışan servisler sırasıyla birbirlerinin cevaplarını bekleyerek ilerlemektedir. Veri akışı sırasında bir servis başka bir servise istek attığında, o servisten cevap gelmesini bekler ve cevap geldikten sonra sonucun başarılı ya da başarısız olması durumuna göre yoluna devam eder.

Bu yöntemde, servisleri yöneten yönetici bir servis ile alttaki servislerin kontrol edilmesi sağlanabilir. Bu yönetici servisin kullanılmadığı durumda veri bütünlüğünü korumak, veri akışını takip etmek çok fazla mikroservis kullanıldığı taktirde zorlaşabilmektedir. Örneğin bir yönetici servisin varlığı altında çalışan mikroservisler birbirleri ile haberleşme yapmayıp, sadece bu yönetici servis ile iletişimde olmaktadır ve yönetici servis kullanıcı ile irtibatta olup işlemi tamamlamak için veri akışını alttaki mikroservislerde yönetmektedir. Bu mimari Şekil 5'te görselleştirilmiştir.



Şekil 5. Yönetici Servis Altında Çalışan Mikroservisler

Yönetici servis bu mimaride bir API geçiti (gateway) olarak kullanılabilmektedir. API geçiti, kullanıcıdan gelen istekleri karşılayıp, ilgili isteği alttaki mikroservislere doğru bir şekilde iletmek, gerekli servisi çağırmak gibi işlemleri yapmaktadır. Aynı zamanda kimlik doğrulama işlemini de API geçiti yapabilmektedir (Yang, Gao, He, ve Li, 2021). Bazı durumlarda ise sadece API geçiti olarak değil, bir veri yöneticisi olarak da kullanılabilmektedir. Bu durumda ise gelen veriyi alttaki mikroservislere dağıtmakla beraber, mikroservislerden gelen verileri toplayıp bir bütün halinde cevap olarak kullanıcıya sunmaktadır. Örneğin, kullanıcı bilgilerini ve satın alma bilgilerini toplayıp cevap olarak dönecek bir sistem tasarlanacak ise, yönetici servis önce kullanıcı bilgilerini kullanıcı servisinden alır, ardından satın alma servisinden bu kullanıcının satın almış olduğu ürünler bilgisini alır ve bunları bir bütün haline getirip kullanıcıya cevap döner. Bu örnekte yönetici servis sadece verileri birleştirip bir bütün haline getirmiştir. Daha detaylı bir sistemde ise yönetici servis verileri toplayıp bunları uygun bir biçimde işleyip kullanıcıya dönüş de yapabilir. Bu durumda veri manipülasyonu yapabilen daha detaylı bir yönetici servis elde edilmiş olur. Bu seviyede kullanılan bir sisteme örnek verecek olursak, satın alınan ürünler ile bunların birim fiyatı üzerinden hesaplayıp kullanıcıya dönüş yapan bir sistem düşünülebilir. Mesela, yönetici servis önce kullanıcının satın almış olduğu ürünlerin bilgisini ürün servisinden aldıktan sonra, bu ürünlerin fiyatlarının tutulduğu başka bir servisten fiyat bilgilerini alır ve daha sonra alınan ürünlerin toplam

fiyatını oluşturup kullanıcıya dönebilir. Ancak bu durumda yönetici servise API geçiti rolünün dışında işlem yapma maliyeti eklendiği için, yönetici servisin tutarlılığı, hata toleransı düşebileceği göz önünde bulundurulmalıdır.

Yönetici servis olmaksızın da mikroservisler birbirlerini direk çağırabilmektedir. Bu durumda hangi işlemin hangi işlemden sonra geleceğini yönetmek zorlaşmaktadır. Çünkü bir sistemde birden fazla işlem yapılır ve her işlemin sırası farklı olabilmektedir. Örneğin, A, B, C, D olmak üzere dört adet mikroservisin olduğu bir sistemde, bir işlem A, B, C, D servislerinin çağırılma sırası ile işlem yaparken, başka bir işlem C, A, D, B sırası ile yapılabilir. Böyle bir durum söz konusu olduğunda, kullanıcıdan ilk isteği hangi servisin alacağı ve hangi servisin kullanıcıya cevap vereceği kararı zorlaşmaktadır. Aynı zamanda yazılım geliştiricilerin sistemi test etmesi, tasarlaması zorlaşmaktadır.

2.4.1.1 REST mimarisi

REST (Temsili Durum Transferi) mimarisi, bir yazılım mimari stili olarak Roy Fieldings'in doktora tezinde ortaya çıkmıştır (Fielding, 2000). Direk servis çağırma yöntemi olarak REST mimarisi örnek verilebilir. REST mimarisinde genellikle HTTP metotları kullanılır. Bu HTTP metotlarından bazıları, GET, bir veriyi çekme isteği, PUT ve POST bir veriyi ekleme isteği, DELETE ise bir veriyi silme işlemini temsil etmektedir. Bu metotlarla, REST mimarisi bir istemci ve sunucu arasındaki iletişimi sağlamaktadır. İstemci yapacağı işlemi bu metotlar yardımıyla servisin ilgili URI'ına gönderip servisin tetiklenmesini sağlar. REST mimarisine sahip sistemlere RESTful sistemler de denmektedir (Kocaman, 2018).

Bir sistemin RESTful olabilmesi için 6 adet kurala uyması gerekmektedir (Costa, Pires, Delicato, ve Merson, 2014). Bunlar, *"İstemci, sunucu mimarisi olması"*, *"Durumsuz olması"*, *"Önbellek kullanımı"*, *"Tekil düzenli arayüzlü olması"*, *"Katman sistemi olması"*, *"İstek üzerine kod eklenebilirliği"*dir (s. 105).

İstemci, sunucu mimarisi olması, istemci ile sunucunun tamamen ayrı olması demektir. Bunun avantajı, bağımsız bir şekilde istemcinin (bir Web önyüz projesi) ve sunucunun (bir mikroservisin) yönetilip geliştirilmesidir.

Durumsuz olması, istemci ve sunucu arasındaki mesajlaşma süresinde, mesajların durum bilgisinin istek ve cevap mesajında olması demektir. Yani karşılıklı iletişimde olan istemci ve sunucunun yürüttüğü iletişimin durum kontrolü sunucuda değil, istemcide olması demektir. Böylelikle sunucu birden çok fazla istemci ile hataya yer olmadan aynı anda farklı iletişimlerde bulunabilir.

Önbellek kullanımı, sunucunun önbellek kullanması gerekliliğidir. Böylelikle sistem daha hızlı ve verimli çalışabilir. Dolayısıyla istemci ve sunucu arasındaki ve sunucu ile veri tabanı arasındaki ağ trafiği de mümkün olabildiğince azalmış olmaktadır.

Tekil düzenli arayüz olması, bir sistemin REST olması için en önemli kurallarından birisidir. Sistemin belli bir şablonda belli bir düzende çalışmasını sağlar. Tekil düzenli arayüz kuralı, kendi içerisinde bir takım alt kuralları getirmektedir. Bu alt kurallardan biri sistem URI kullanarak istemcinin hangi işlem kümesini kullanacağı URI bazında sunucu üzerinde ayrılmış olmalıdır. Örneğin, kullanıcı işlemleri yapan bir sistemde, sunucu adresinden sonra */kullanici* URI ile istemci kullanıcı oluşturabilir, kullanıcı silebilir, kullanıcı sonuçlarını listeleyebilir. Diğer bir alt kural ise, sistemin HTTP GET, POST, PUT gibi metotlarını kullanmasıdır.

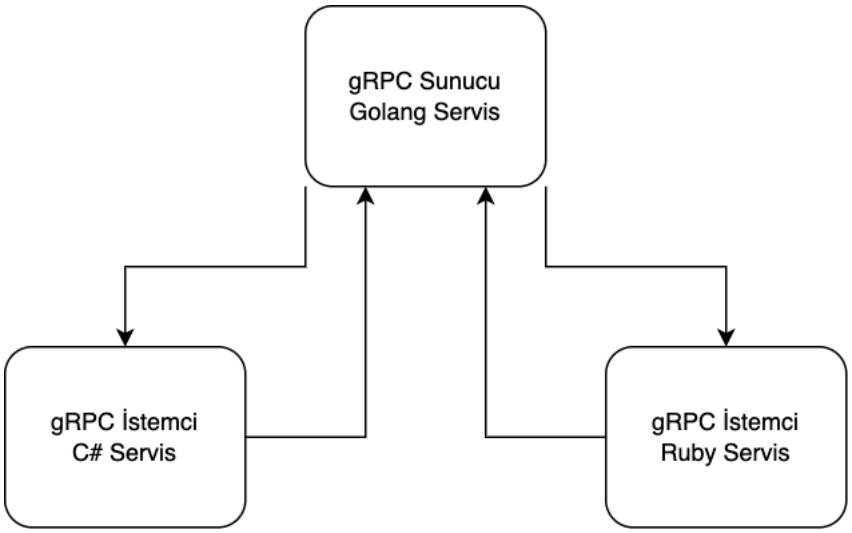
Katman sistemi olması, sistemin, genellikle sunucunun kendi içerisinde servis katmanlarına bölünmüş yapıda tutulmasını söyler. Örneğin bir servis kendi içerisinde servis katmanı, iş katmanı gibi katmanlara sahip olabilir. Aynı zamanda kullanılan bir yük dengeleyici de servisin çalışmasından önce tetiklenip servisin erişilebilirliğini artırarak bir katman olarak düşünülebilir. İstemci bu yük dengeleyiciye istek atar, yük dengeleyici arkada çalışan servislerden uygun olana isteği yönlendirir. Böylelikle katman sistemi kurulmuş olur.

İstek üzerine kod eklenebilirliği kuralı hem istemci tarafında hem de sunucu tarafında gelen iş taleplerini karşılayacak şekilde kolaylıkla yeni bir servis yapısının eklenmesidir. Sistem bu sayede hantal ve kötü çalışan bir sistem olmaktan ziyade, yeni iş yapılarına uygun, istenildiği zaman kod eklenip çıkartılabilinen güvenilir ve güncel bir yapıda olur.

2.4.1.2 Uzaktan yordam çağrısı (RPC)

Bir istemcinin sunucudaki bir fonksiyonu ağ aracılığı ile uzaktan çalıştırmasına uzaktan yordam çağrısı (RPC) denmektedir. REST mimaride bir istemci, sunucudaki bir metodu çağırmak için sunucuya HTTP isteği atmaktadır. Ancak RPC'de istemci sunucudaki bir fonksiyonu ağ ile bağlı olduğu için direk kendi istediği girdiler ile çalıştırabilir (Golub, Jakobović, ve Janeš, 2005).

RPC teknolojisi uzun yıllarca kullanılmıştır. Daha sonraları Google firmasının geliştirdiği RPC uygulaması olan gRPC günümüzde sıklıkla kullanılmaktadır. gRPC genellikle protokol tamponu (protocol buffer) kısaca protobuf kullanmaktadır (gRPC İnternet Sitesi, 2022b). Protobuf Google'ın geliştirdiği yazılım dili bağımsız, açık kaynaklı bir veri formatlama dosya tipidir (Protocol Buffers İnternet Sitesi, 2022). Şekil 6'da görüldüğü gibi, gRPC ile sunucu ve istemci karşılıklı olarak birbirlerini çalıştırabilirler. Bu istemci ve sunucular farklı yazılım dillerinde de yazılmış olabilir.

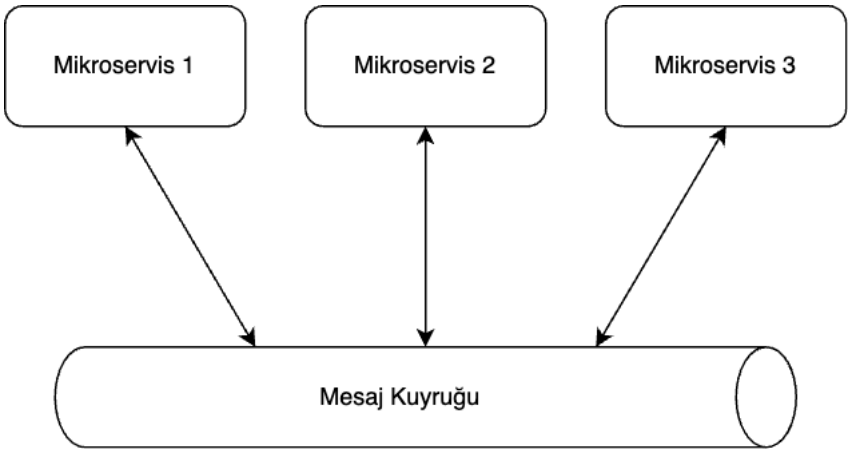


Şekil 6. gRPC Sunucu ve İstemci Çağırma İşlemi

2.4.2 Kuyruk tabanlı servis çağırma yöntemi

Bu yöntemde servisler birbirlerini senkron olarak değil, asenkron olarak bir kuyruk sistemi ile çağırmaktadır. Böylelikle servisler birbirlerini beklemeden işlemlerini tamamlayıp sıradaki işleme geçebilmektedir. Bu yöntemde her servis bir kuyruğa abone

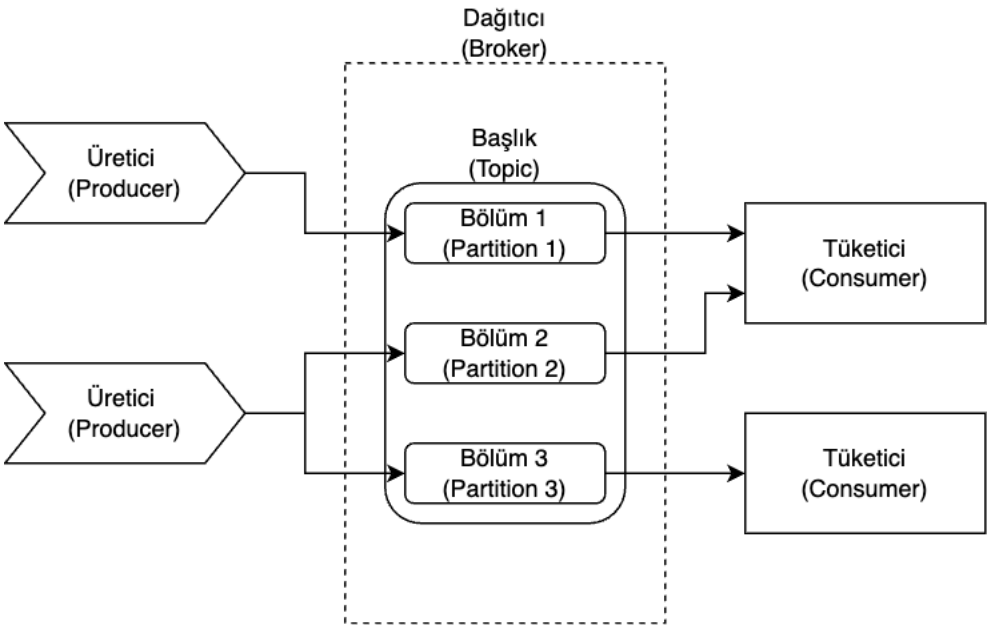
olup, kendisi için kuyruğa gönderilmiş bir mesajı okuduklarında işleme başlayıp, işlemi tamamladıklarında ise kuyruğa işlemi başarılı ya da başarısız bitirdiğine dair bir mesaj göndermektedir. Başka bir mikroservis de bu mesaja göre kendi işlemini yapmaktadır. Bu mimari Şekil 7'de görselleştirilmiştir.



Şekil 7. Mesaj Kuyruğu ile Çalışan Mikroservisler

Bir mesaj kuyruğu, mikroservislerin hem hata hem de başarılı sonuçlarının depolandığı dağıtık yapıya sahip olabilen bir platform olarak tanımlanabilir (Daraghmi, Zhang, ve Yuan, 2022). Dağıtık mimariye sahip mesaj kuyruğu olarak Apache Kafka kullanılabilir. Kafka, açık kaynaklı dağıtık olay akış platformudur. Mikroservisler bu mesaj kuyruğuna veri gönderebilir, aynı zamanda bir kuyruğa abone olabilir. Kafka'nın en önemli özelliği yüksek trafik altında stabil çalışması ve saniyede binlerce mesajı işleyebilmesi olarak sıralanabilir (Kreps, Narkhede, ve Rao, 2011). Linkedin, Yahoo, Twitter gibi birçok büyük teknoloji firması Kafka'yı kullanmaktadır. Kafka sadece bir mesaj kuyruğu olarak değil, aynı zamanda log izleme, hata takibi, web site aktivitesi gibi konularda da kullanılmaktadır (Sharvari ve Sowmya Nag, 2019). Kafka'nın mimarisi genel olarak, bölmeler (partition), dağıtıcı (broker), başlık (topic), üretici (producer) ve tüketici (consumer) olarak sıralanabilir. Başlıklar mesajların tutulduğu kullanıcının isimlerini verdiği yerdir. Başlıklar birden fazla bölmelerden meydana gelmektedirler. Dağıtıcılar ise bu başlıkların tutulduğu sunuculardır. Birçok dağıtıcı bir araya gelip Kafka kümelerini

oluşturur. Üreticiler ise mesajları üreten ve tüketiciler ise mesajları tüketen servislerdir (Kreps, Narkhede, ve Rao, 2011). Kafka'nın kuyruğa veri basım ve kuyruktan veri alım işlem şeması Şekil 8'te gösterilmiştir.

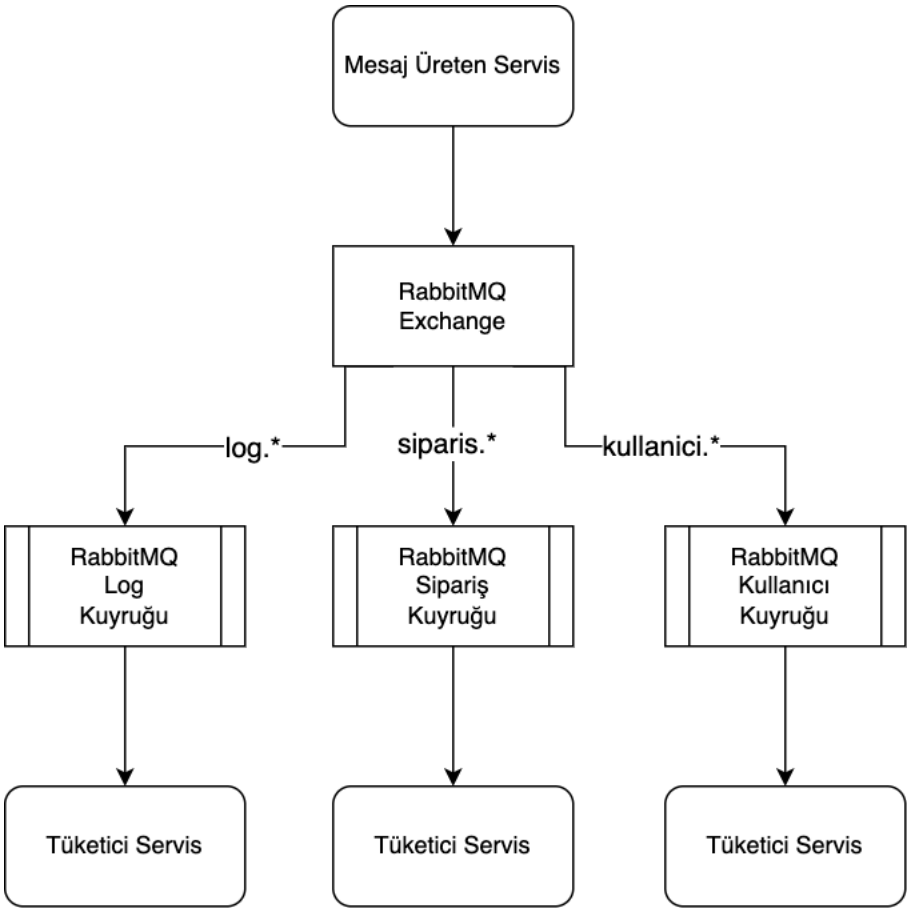


Şekil 8. Kafka Üretim Tüketim Mimarisi

Apache Kafka'nın dışında piyasada çok kullanılan bir diğer kuyruk mesaj platformu olarak RabbitMQ sayılabilir. RabbitMQ da Kafka gibi açık kaynaklı bir kuyruk yönetim platformudur ve bir AMQP (Advanced Message Queuing Protocol) platformudur. RabbitMQ ile Kafka kıyaslanacak olunursa, Kafka dağıtık mimariye sahip bir kuyruk yönetim biçimidir ve ek olarak, Kafka'nın kuyruğundaki veriler direk sabit diske kaydolduğu için kalıcıdır. RabbitMQ'da ise veriler belleğe kaydolmaktadır, dolayısıyla veriler kalıcı değildir. Ancak RabbitMQ'da aynı zamanda verileri sabit diske kaydetme özelliği aktive edildiğinde kuyruktaki mesajların kalıcılığı sağlanabilir (Sharvari ve Sowmya Nag, 2019) (Dobbelaere ve Esmaili, 2017).

RabbitMQ ile oluşturulmuş bir kuyruk yapısı Şekil 9'da görülmektedir. Mesaj üreten bir servis RabbitMQ'nun exchange başlığına veriler göndermektedir. RabbitMQ exchange

ile mesajın başlığı ile farklı kuyruklara yönlendirme yapılabilmektedir. Bu örnekte kuyruktaki mesajlar 3'e ayrılmıştır. Bunlar log, sipariş ve kullanıcı mesajlarını içeren kuyruklardır. Örneğin bir kullanıcı oluşturma isteğini mesaj üreten servis kuyruğa göndermek istediğinde kullanici.123 gibi bir isimle mesajı exchange'e gönderdiğinde RabbitMQ bunu otomatik algılayıp kullanıcı kuyruğuna bu mesajı iletir. Ardından kullanıcı oluşturan bir tüketici servis bu veriyi alıp kullanıcı oluşturmak için gerekli işlemleri uygular. Sipariş ve log kuyrukları da aynı şekilde çalışmaktadır.



Şekil 9. RabbitMQ Exchange Kuyruk Yapısı

Kuyruk tabanlı sistemde, direk çağırma yöntemine göre yönetici servis ihtiyacı daha az olmaktadır. Çünkü, servisler birbirlerini beklemeden ve kendisinden sonra hangi servisin geleceğini düşünmeden işlemlerini yapar ve işlemlerinin sonucunu ilgili kuyruğa basar. Ancak bazı durumlarda yönetici bir servis de kullanılabilmektedir. Saga yöntemi bu

kuyruk yapısını baz alarak sistem tasarımı sunar. Hem yönetici servisin varlığı ile hem de yönetici servisi olmadan çözüm sunmaktadır.

Kuyruk sisteminde, geri alma senaryolarını işletmek ve yönetmek, direk servis çağırma yöntemine göre daha kolay yapılabilmektedir. Çünkü, her bir servisin işleminin başlaması için kuyruğa bir mesaj gönderilmesi gibi, geri alma işlemi için de kuyruğa o servis için mesaj gönderilmesi yeterlidir. İlgili servis de bunu geri alma işlemi olarak işleyip sistemi eski haline çevirmektedir. Ancak burada dikkat edilmesi gereken husus, hata alınan noktadan önce işlemini yapıp bitirmiş servislere geri alma mesajını gönderecek olan servisin doğru ve dikkatli bir biçimde geliştirilmesi gerekmektedir. Çünkü, geri alma işleminde kuyrukta ya da herhangi bir serviste oluşacak bir hata sistemin veri tutarlılığını tamamen bozabilir. Bunun önüne geçebilmek için, sistemde var olan tüm mikroservislerde çıkabilecek bir sorun için tek tek geri alma senaryoları düşünülüp, uygun bir şekilde kuyruk mesajları oluşturulmalıdır.

2.5 Mikroservis Mimarisinde Dağıtık İşlemlerin Yönetimi

Bir yazılım projesinde sistem genellikle tek bir servisten oluşmaz. Birden çok işlemlerin, servislerin bir araya gelmesi ile bir sistem oluşur. Örneğin, bir e-ticaret sitesinde, kullanıcıya sadece bir ara yüz verilip sanki tek bir sistem var gibi gözükse de arkada var olan servisler, stok işlemlerinin yürütüldüğü servis, kullanıcıların kontrol edildiği servis, siparişlerin kontrol edildiği servis gibi pek çok alt servislerden oluşur. Bu tarz çoklu servislerin yönetimi dağıtık işlemlerin yönetimine örnektir. MM'de tüm kod tek bir çatı altında olduğu için, dağıtık işlemler projedeki diğer dosyalarda yürütülür ve yönetimi kolaydır. Ancak, asenkron işlem yapmak mümkün değildir, çünkü sistem tek bir çalıştırılabilir çıktı ürettiği için akış sabit olacaktır. MSM'de ise asenkron dağıtık işlemlerin yapılması mümkündür, çünkü servisler birbirinden bağımsız çalışmaktadır.

Veri tabanı işlemlerinde, bir işlemin güvenilirliğini garanti altına almak için ACID prensibi ortaya atılmıştır (Medjahed, Ouzzani, ve Elmagarmid, 2009). Bu prensip, atomiklik, tutarlılık, izolasyon, sağlamlık olmak üzere dört başlık altında toplanır.

- Atomiklik: Birden çok veri tabanı bir işlem ile güncelleniyorsa, bu veri tabanları ya tamamen başarılı sonuç verecek ya da hata almalıdır. Tutarsız bir durum olmamalıdır.
- Tutarlılık: Genel olarak veri bütünlüğü anlamına gelmektedir. Bir sistemde bir işlem devam ederken, sisteme bir istek geldiğinde veriler tutarsız bir şekilde dönülmemelidir. Arkada işlem devam ederken bile sistem tutarlı sonuç dönmelidir.
- İzolasyon: Birbirinden ayrı sistemlerin birbirinden ayrı bir şekilde işlemlerini yürütebilme kabiliyetidir. Sistemdeki mikroservisler her ne kadar beraber çalışsalar da işlemlerini bağımsız ve izole bir şekilde yürütebilmelidirler.
- Sağlamlık: Sistemde bir işlemin akışı sırasında hata oluşur ise, sistem kendini bir önceki haline sorunsuz bir şekilde döndürmelidir. Hata ile karşılaşan sistemin kendi kendini eski haline getirme kabiliyetidir (s. 1221).

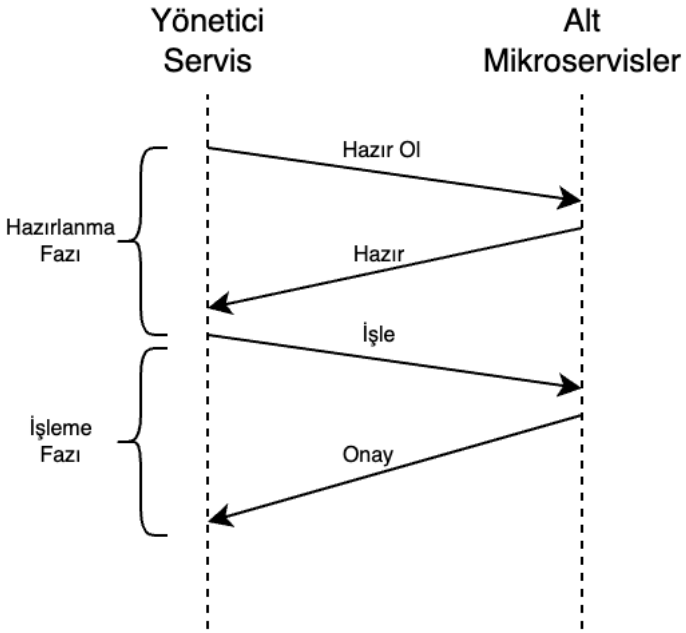
MSM'de dağıtık işlemlerin yönetimini ele alan yöntemler genel olarak iki ana başlık altında toplanabilir. Bunlardan biri iki fazlı işleme (Two Phase Commit, 2PC) diğeri de saga yöntemidir. Bu iki yöntem de ACID prensiplerini karşılamak için kullanılabilir. Ancak hem saga yöntemi hem de 2PC yöntemi ACID prensiplerini tamamen karşılayamayabilir (Stefanko, Chaloupka, ve Rossi, 2019) (Hohpe, 2005).

2.5.1 İki fazlı işleme (2PC)

2PC atomik işleme protokollerinin en basit olanı ve en çok çalışılan olanıdır (Al-Houmaily ve Samaras, 2009). 2PC protokolü uzun yıllardır kullanılan bir yaklaşım olduğu için birçok farklı varyasyonu vardır (Samaras, Britton, Citron, ve Mohan, 1993). Bir sistemde 2PC yöntemi kullanılacak ise alt mikroservisleri yöneten bir koordinatör servisin bulunması gerekmektedir. Bu koordinatör servis işlem yapacak olan bu alt mikroservislere önce hazır ol mesajı gönderip, eğer tüm mikroservisler hazır ol mesajına olumlu cevap veriyorlar ise, koordinatör servis daha sonra işlemi yap komutu gönderir.

Yönetici servis alt servislere ilk önce hazır ol komutu gönderdikten sonra tüm servislerden gelecek olan onay mesajını beklediği için, en az bir adet hazır olmayan servis var ise işlem yap komutu gönderilmeyeceği için, veri tutarlılığı bu şekilde sağlanır. Aynı şekilde tüm servisler hazır olduğuna dair mesaj gönderirse, ardından işlem yap komutu

ile tüm servislere bu veri düzgün bir şekilde işleneceği için veri tutarlılığı sağlanmış olur (Al-Houmaily Y. , 2010). Ancak işlem yap komutunda bir hata olur ise, sistemin eski haline gelmesi için yönetici servisin bir geri alma senaryosu uygulayıp verileri işlem yapılmadan önceki haline alması gerekmektedir. 2PC işlem yönetimi Şekil 10'da gösterilmiştir.



Şekil 10. 2PC Çalışma Mekanizması

2PC'nin avantaj ve dezavantajları şu şekilde sıralanabilir.

Avantajları:

Hazır ol ve işle fazları birbirinden ayrı olduğu ve işleme fazı için hazır ol komutu tamamlandığı için veri tutarlılığı sağlanmış olur.

Yönetici servisin varlığı ile tek bir kullanım noktası olacağı için kullanım kolaylığı sağlamış olur.

Dezavantajları:

Yönetici servis alt mikroservislere bekle komutu gönderdiği zaman, tüm servislerden gelecek onay mesajını beklerken sistem kilitlenmiş olur.

Alt mikroservislerde işleme adımında hata olur ise, detaylı bir geri alma senaryosu düşünülüp tüm diğer servislere uygulanmalıdır.

Tek bir yönetici servisin bulunması avantaj olduğu kadar dezavantaj olabilir, yönetici serviste oluşacak bir hata tüm sistemi çalışmaz hale getirebilir.

2.5.2 Saga yöntemi

Saga yöntemi, bir işi yapmak için bir dizi operasyonel işlemin yapılması olarak özetlenebilir (Garcia-Molina ve Salem, 1987). Saga ana işlemi birden çok küçük işleme bölerek mikroservislerin işlem yapmasını sağlar. Mikroservisleri bu sırada kilitlemez, gelen işlemi yapıp sıradaki aşamaya geçer (Dürr, Lichtenthaler, ve Wirtz, 2021). Saga yönteminde bir işlem birden çok alt işlem olarak bölünüp mikroservislere dağılmıştır. Saga yönteminde çalışan alt mikroservisler işlemi yapmak için ilgili kuyruktan başlama mesajını alıp daha sonra işlemini bitirdiğinde, başarılı ya da başarısız olması durumunda ilgili bir kuyruğa işlemin sonucunu gönderir. Mikroservisler bu şekilde haberleşip bir iş bütününü tamamlamış olur. Saga yöntemi, bu işlemlerin akışı sırasında bir hata olması durumunda sistemi geri alma senaryoları ile eski haline geri döndürür. Böylelikle tam olarak ACID prensiplerinin tamamını sağlayamamış olsa da atomiklik, tutarlılık ve sağlamlık prensiplerini (ACD) garanti altına alır ve veri tutarlılığını sağlamış olur (Stefanko, Chaloupka, ve Rossi, 2019). Burada Saga yönteminin sağlayamadığı izolasyon terimi, daha tamamlanmamış bir bütün işlemde, veri tabanlarına yazma ve okuma işleminin yapılabilmesinden kaynaklanmaktadır (Daraghmi, Zhang, ve Yuan, 2022).

Saga yönteminin amacı da 2PC gibi dağıtık işlemlerde veri tutarlılığını sağlamaktır. Bunu yaparken 2PC'deki gibi bir yönetici servis ile hazır ol ve işle komutu yerine, her bir servisin kuyruktan mesajı alıp kendi işlemini yapmasını sağlar. Bunu yapabilmek için saga yönteminde orkestarsyon ve koreografi yaklaşımları olarak iki farklı yaklaşım vardır. Orkestrasyon yaklaşımında bir yönetici servis bulunurken, koreografi yaklaşımında servisler birbirleri ile bir kuyruk yapısı ile direk haberleşmektedirler (Daraghmi, Zhang, ve Yuan, 2022).

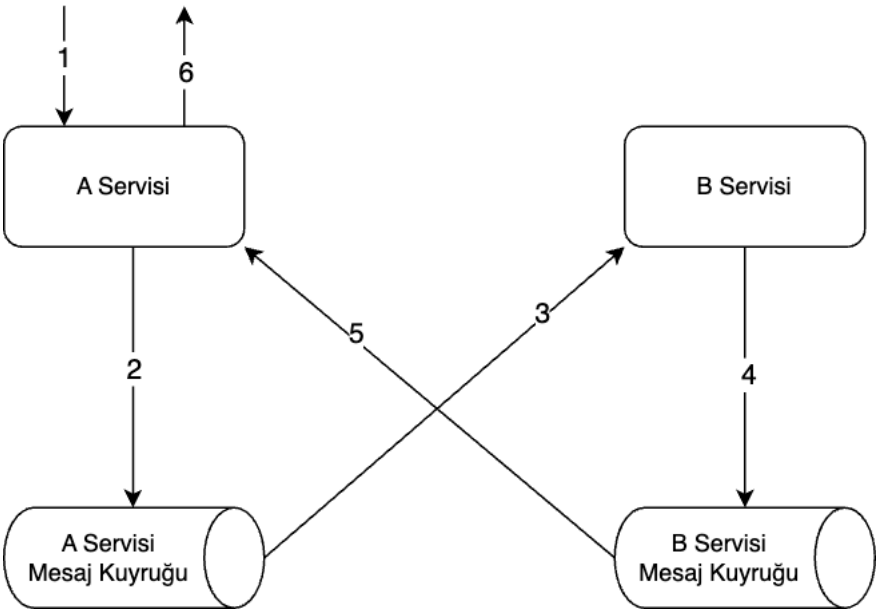
2.5.2.1 Koreografi yaklaşımı

Koreografi yaklaşımında mikroservisler birbirleri ile kuyruk sistemi ile haberleşmektedirler. Bunları yöneten ayrıca bir yönetici servis bulunmamaktadır. Her bir servis dinlediği kuyruktan başlaması gerektiği bilgisini alarak işlemi yapar ve ardından kuyruğa işlemi başarılı ya da başarısız bitirdiğine dair bilgiyi gönderir. Başka bir mikroservis de bu bilgiye göre aynı şekilde işlemini gerçekleştirir (Rudrabhatla, 2018).

Koreografi yaklaşımı ile bir sistem tasarlanacak ise, öncelikle alt mikroservislerin hangi durumda (kuyruktaki hangi mesaj ile) işlemine başlayacağını belirlemek gerekmektedir. Ardından kuyruğa hangi mesajı atacağını dolayısıyla hangi mikroservisin işleme başlayacağını belirlemek gerekmektedir. Bu şekilde tasarlanan mikroservisler birbirleri ile direk ilişkide olmasa bile, aynı kuyruktaki farklı mesajları dinleyerek iletişimde olup bir bütün işlemi gerçekleştirmiş olurlar. Çok fazla mikroservisin bulunduğu bir sistemde koreografi yaklaşımı ile saga uygulanır ise, bir işlemin hangi sıra ile çalışacağını belirlemek zor olabilir. Çünkü mikroservisler birbirleri ile iletişimde değil, kuyruktaki mesajları dinleyerek iletişim kurmuş olurlar. Bu da çok fazla mikroservisin olduğu durumlarda yazılım geliştiricinin kafasının karışmasına sebep olabilir.

Bu yaklaşım ile geliştirilen mikroservislerde esas olarak iki adet işlem yönelimi vardır. Bunlardan biri devreye alma işlemi, diğeri de geri alma işlemi olarak sayılabilir. Bir mikroservis kuyruktan kendi işleminin başlayacağına dair bir mesaj dinler ayrıca olası bir hata durumunda geri alma işlemini başlatacak olan bir diğer kuyruk mesajını dinlemektedir. Bunların dışında işlemin bittiğini temsilen tek bir kuyruk dönüş mesajı oluşturur. Bu mesaj ya başarılı ya da başarısız bittiğine dair bir bilgi içerir. Eğer ki işlem başarısız olur ise bu mikroservis kendi yaptığı işlemleri geri alıp daha sonra işlemin başarısız bitirildiğine dair bilgiyi kuyruğa gönderir.

Burada dikkat edilmesi gereken bir diğer husus da tüm mikroservisler işlemlerini bitirdikten sonra bütün işlemin başarılı ya da başarısız bittiğini kullanıcıya bildirmek ya bir mikroservisin görevi altında olabilir ya da mikroservislerin çıktılarını dinleyen ayrı bir servis tüm işlemin başarılı ya da başarısız bittiğini teyit edip kullanıcıya bilgi verebilir. Şekil 11'de ilk tetiklenen mikroservis ile kullanıcıya dönüş yapılan iki mikroservisin bulunduğu bir sistemin başarılı sonuç üreten bir görseli yer almaktadır.



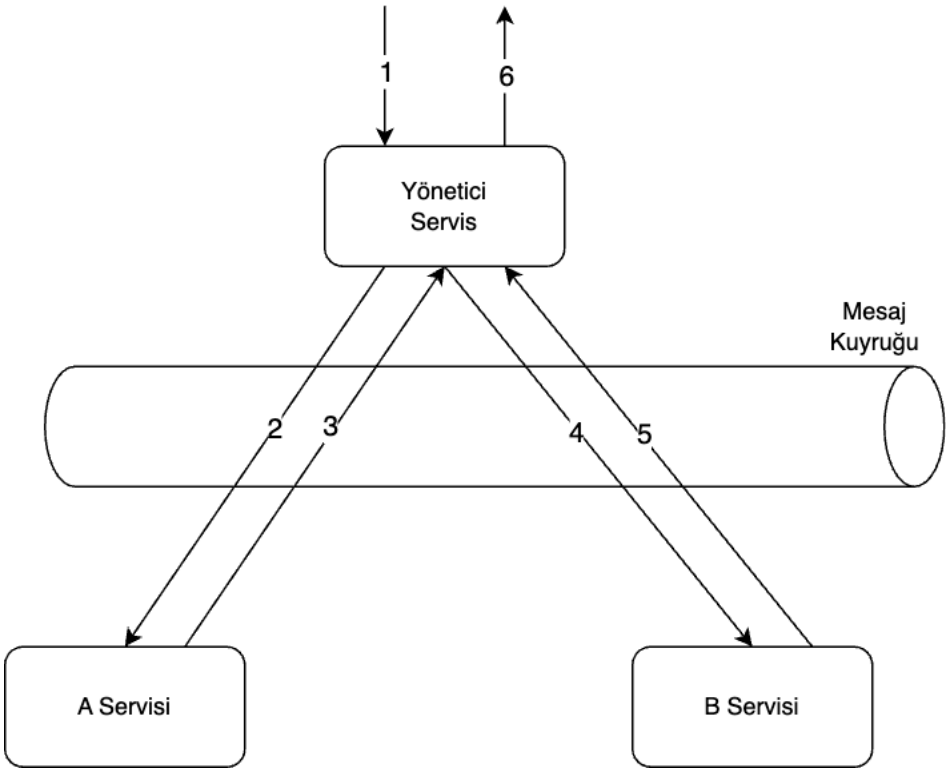
Şekil 11. Başarılı Sonuç Dönen Saga Koreografi Yaklaşımı

Şekil 11'de görüldüğü gibi, 1. adımda A servisi kullanıcı tarafından tetiklenmektedir. 2. adımda A servisi işlemini başarılı bitirdikten sonra kuyruğa A servisinin işleminin başarılı bittiği bilgisine sahip mesajı iletmektedir. 3. adımda B servisi bu mesajı alıp kendi işlemine başlamaktadır. 4. adımda B servisi kendi işlemini başarılı bitirip bu bilgiyi kuyruğa iletmektedir. 5. adımda B servisinin ilettiği başarılı kuyruk mesajı A servisi tarafından okunup tüm işlemlerin başarılı bittiği sonucuna ulaşıp tüm servislerin ve dolayısıyla tüm işlemlerin başarılı bittiğine dair bilgiyi 6. adımda kullanıcıya geri iletmektedir. 6. adımda işlemin başarılı bittiğine dair bilgiyi A servisi bir REST servisi çağırarak ya da başka bir kuyruğa bu mesajı ileterek yapabilir.

2.5.2.2 Orkestrasyon yaklaşımı

Orkestrasyon yaklaşımında alttaki mikroservisleri kontrol eden yönetici bir servis vardır. Koreografi yaklaşımından farkı bu yönetici servisin bulunmasıdır (Rudrabhatla, 2018). Aynı şekilde bu yaklaşımda da kuyruk yapısı kullanılabilir. Yönetici servis alttaki mikroservisler ile kuyruk sistemindeki mesajlar ile iletişim kurar ve bunları yönetir. Alt mikroservislerden biri işlem akışı sırasında bir hata alır ise bu hatayı kuyruk sistemine mesaj olarak iletip yönetici servisin bu mesajı almasını sağlar. Yönetici servis bu hata

mesajını aldığı zaman bir geri alma işlemi başlatır ve işlem yapmış olan önceki mikroservislerin işlemini geri almasını sağlar. Bu işlem de aynı şekilde kuyruk haberleşme yöntemi ile olmaktadır. Burada geri alma işlemini yapan yönetici servis değil, yönetici servisin tetiklediği ilgili mikroservislerdir. Şekil 12'de yönetici servisin kontrolü altında kuyruk haberleşme ile iletişimde olan iki mikroservisin var olduğu başarılı sonuç üreten bir orkestrasyon saga yaklaşımı yer almaktadır.



Şekil 12. Başarılı Sonuç Dönen Saga Orkestrasyon Yaklaşımı

Şekil 12'de gösterildiği gibi, 1. adımda yönetici servis kullanıcı tarafından tetiklenir. 2. adımda yönetici servis kuyruk üzerinden A servisini çağırır. 3. adımda A servisi başarılı işlemini yapıp bu mesajı kuyruğa gönderir ve yönetici servis bu mesajı alır. 4. adımda yönetici servis B servisini çağırır. 5. adımda B servisi işlemini başarılı tamamlayıp sonucunu kuyruğa gönderir ve yönetici servis bu mesajı alır. 6. adımda yönetici servis tüm işlemlerin başarılı bittiğini kullanıcıya iletir.

Yönetici servis ile yapılmış bir orkestrasyon saga yaklaşımı hem senkron hem asenkron olarak yapılabilir. Senkron olarak yönetilen bir sistemde, her bir alt mikroservisin işlemini yapıp sıradaki işleme geçmesi beklenir. Ancak asenkron çalışan sistemde yönetici servis alt mikroservisleri paralel olarak çağırıp, tüm işlemlerin sonucunu paralel dinler, olası hata durumunda geri alma işlemini ilgili mikroservisleri çağırarak yapabilir. Alt mikroservisler birbirlerini kuyruk mesajları ile asenkron tetikleyebileceği gibi direk mikroservis çağırma yöntemindeki gibi, bir HTTP isteği ile de yönetebilir.

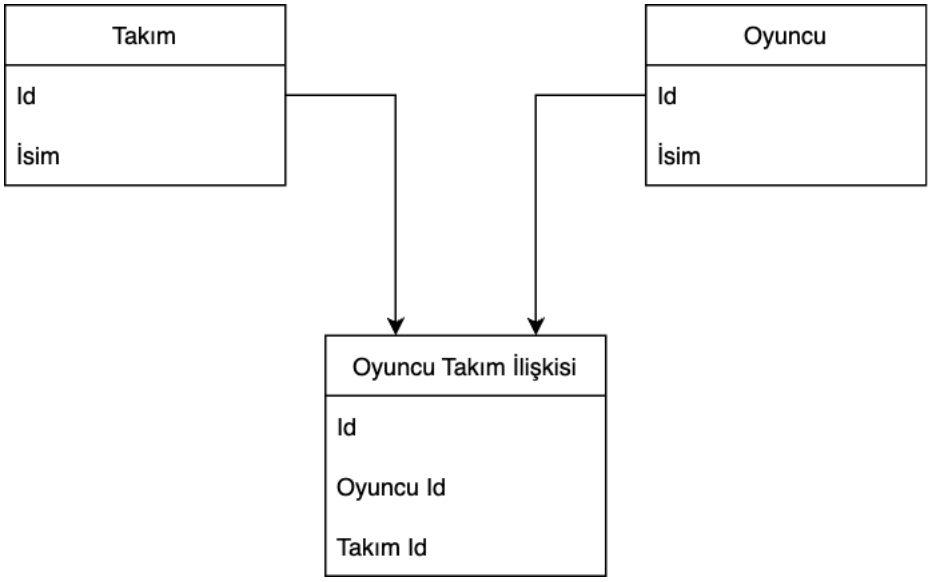
Tek bir yönetici servisin var olması hem avantaj hem dezavantaj sayılabilir. Tek bir yönetim noktası olması kullanım kolaylığı sağlamaktadır. Yani kullanıcı için sadece tek bir ilgili servis vardır. Aynı zamanda yazılım geliştirici için sistemi yönetmesi orkestrasyon yaklaşımında daha kolay olabilir. Diğer yandan, tek bir yönetim merkezi olması olası bir hata durumunda sistemin tamamen çalışamaz hale gelmesine neden olmaktadır. Sistem her ne kadar mikroservisler ile çalışsa da yönetici servis kontrolü altında olan mikroservisler olduğu için yönetici servisteki olası bir hata sistemin durmasına sebebiyet verebilir.

2.6 Geri Alma (Rollback) Senaryoları

Geri alma senaryoları sistemin bütünlüğünü ve tutarlılığını sağlamak için çok önemlidir. Eğer olası bir hata durumunda geri alma işlemleri yapılmaz ise sistem hatalı sonuçlar verebilir. Geri alma işlemleri ile, sistemdeki veriler tutarlı halde kalması sağlanmış olur ve sistem ACID prensiplerini karşılamış olur. Saga ve 2PC yöntemleri ile oluşturulan sistemlerde geri alma senaryoları farklı şekillerde uygulanabilir. Yönetici servis ile yönetilen sistemlerde yönetici servis genel olarak geri alma işlemlerini yönetirken, koreografi yaklaşımı ile oluşturulan sistemlerde geri alma işlemlerini her bir alt mikroservis yönetmektedir.

Yönetici servis altında çalışan alt mikroservislerden birinde bir hata meydana gelir ise, bu bilgi yönetici servise iletilir. Yönetici servis bu hata alan servisten önce çalışmış olan mikroservislerin geri alma işlemini başlatır. Daha önceden çalışmış olan servislerden hangisinin ilk geri alınacağına karar vermek önemlidir. Çünkü bazı servisler birbirlerine bağımlı olabilir. Örneğin, bir servisin çıktısındaki bir veri ile işlem yapıp bu veri ile kendi verisini veri tabanında kayıt yapan başka bir servis var ise, öncelikle bu en son işlemi

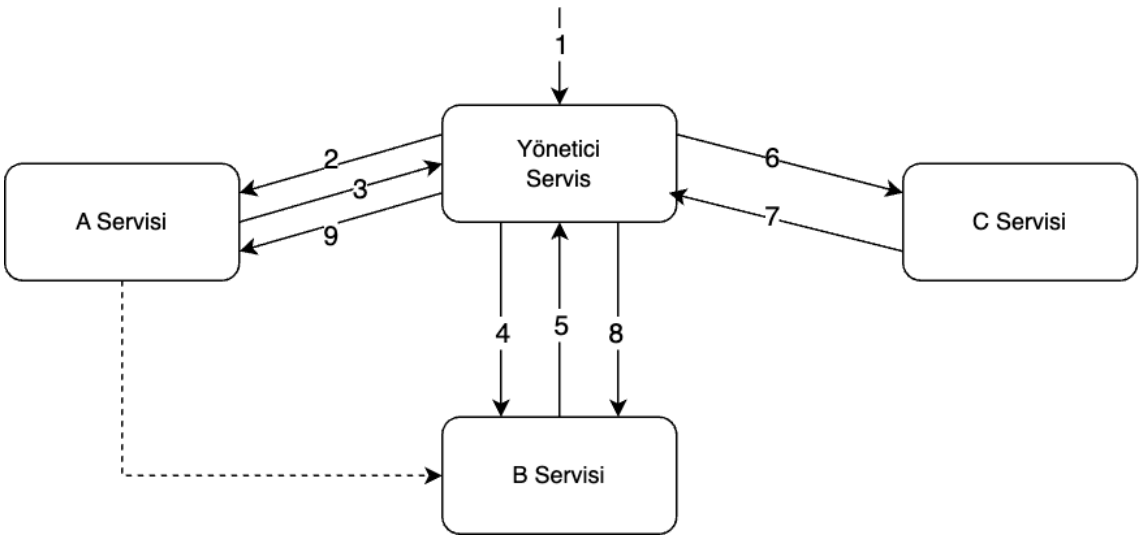
yapan servisin geri alınması gerekmektedir. Aksi durumda sistem olmayan bir veriyi silmeye çalışacağı için hata verebilir. Bu akışı veri tabanı diyagramlarındaki gibi bir diyagram ile görselleştirip geri alma senaryoları görsel olarak çıkartılabilir. Şekil 13'te iki veri tabanı ile bu iki veri tipinin ilişkisi ile oluşturulmuş üçüncü bir tablo görülmektedir.



Şekil 13. Basit Veri Tabanı İlişki Diyagramı

Şekil 13'te görüldüğü gibi Oyuncu, Takım ve bu iki tabloyu birleştiren ve oyuncular ile takımların eşleştirildiği bir tablo görülmektedir. Yeni bir oyuncu oluşturulup, ardından bir takım ile eşleşme yapıldığında öncelikle Oyuncu veri tabanında oyuncu kaydolacaktır. Ardından Oyuncu Takım İlişkisi tablosunda bu ilişkilendirme kaydedilecektir. Olası bir hata durumunda ise, önce Oyuncu Takım İlişkisi tablosundaki kayıt silinmeli ardından da Oyuncu tablosundan ilgili oyuncu silinmelidir. Aksi taktirde Oyuncu Takım İlişkisi tablosunda var olmayan bir oyuncunun kaydı olacaktır ve bu kayıt silinmeye çalışıldığında hata verebilir. Daha kompleks yapılarda ise veri tabanı diyagramındaki en uç noktadaki işlemin geri alınması ile başlanıp yukarıya doğru geri alma adımları devam etmelidir. Bu akışı yönetici servis yönetmelidir ve bu sıralamanın nasıl yapılacağı da yönetici servis içerisinde kodlanmalıdır.

Şekil 14'te orkestrasyon yaklaşımı ile yapılmış A, B ve C mikroservislerini içeren bir sistemde hata meydana gelmesi halinde yürütülecek adımların görseli yer almaktadır. Burada B servisi, A servisinin çıktısı ile elde edilen bir veriyi kullandığı için B servisi A servisine bağımlı durumdadır. Bu durum geri alım sırasında B servisinin A servisinden daha önce geri alım yapılması gerektiğini ortaya koymaktadır. 1. adımda yönetici servis kullanıcıdan isteği alarak işlemi başlatır. 2. adımda A servisini çağırır. 3. adımda A servisi işlemini başarılı bitirip bilgiyi yönetici servise iletir. 4. adımda yönetici servis B servisini çağırır ve 5. adımda B servisi başarılı sonucunu yönetici servise döner. 6. adımda C servisi çağırılır ve 7. adımda C servisi işlemini başarısız bitirdiği bilgisini yönetici servise döner ve yönetici servis geri alım işlemlerini başlatır. 8. adımda B servisi için geri alma komutu gönderilir. Bu işlem tamamlandığında 9. adımda A servisine geri alma komutu gönderilir. Sonuç olarak yönetici servis işleme başlamış, ama işlemler sırasında hata ile karşılaşmış ve yapılmış işlemleri başarılı bir şekilde geri almış ve sistemin tutarlılığını sağlamış olmaktadır.

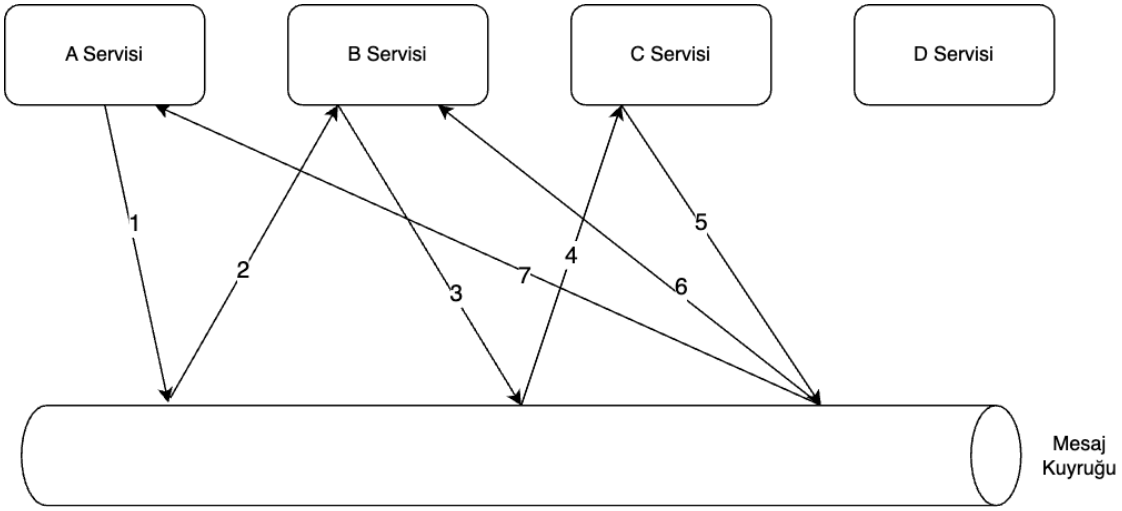


Şekil 14. Orkestrasyon Yaklaşımında Geri Alma İşlemi Akış Şeması

Koreografi yaklaşımı ile yapılan sistemlerdeki geri alma senaryoları orkestrasyon yaklaşımdakini göre daha karmaşık olabilmektedir. Çünkü yönetici bir servis altında geri alma senaryoları tek bir kod bütününde daha rahat yönetilebilir. Koreografi yaklaşımında işlem akışı sırasında alt mikroservislerden birinde bir hata oluşur ise hata oluşan

mikroservis mesaj kuyruğuna işleminin hatalı bittiğine dair bir mesaj gönderir. Bunu dinleyen ve daha önce işlem yapmış olan mikroservisler bu mesajı alıp geri alma senaryolarını kendileri yönetecek şekilde başlatırlar. Geri alma işlemi yapan tüm servisler işlemlerini bitirdiklerinde bu tüm alt mikroservisleri dinleyip sonuç üreten bir servis aracılığı ile kullanıcıya işlemin başarısız bittiğini, ama geri alma işleminin başarılı bittiği bilgisi verilebilir.

Şekil 15'te koreografi yaklaşımı ile oluşturulmuş A, B, C ve D olarak 4 mikroservis içeren bir sistemdeki olası bir hata durumundaki işlemlerin adımlarını gösteren görsel yer almaktadır. Bu görselde 1. adımda sistem A servisinin çalışması ile işleme başlar ve A servisi işlemini başarılı bitirdikten sonra bu bilgiyi kuyruğa gönderir. 2. adımda B servisi bu bilgiyi kuyruktan alır ve kendi işlemini gerçekleştirdikten sonra 3. adımda başarılı gerçekleşen işleminin cevabını kuyruğa gönderir. 4. adımda C servisi dinlediği bu mesajı yakalar ve kendi işlemine başlar, ama bir hata ile karşılaşır ve işlemini hatalı tamamladığına dair bilgi içeren mesajını 5. adımda kuyruğa gönderir. C servisinden önce çalışmış olan diğer servisler bu hata mesajına abone olmuş durumdalardır ve bu hata mesajını yakalarlar. Bu örnekte A ve B servisi 6. ve 7. adımda bu mesajı alırlar ve daha önce kendi işlemiş oldukları işlemleri paralel bir şekilde geri alırlar. Şekil 15'te görüldüğü gibi tüm işlemler D servisi hiçbir işlem yapmadan bitmiştir. Çünkü D servisine sıra gelmeden bir hata oluşmuş ve geri alma işlemleri başlamıştır.



Şekil 15. Koreografi Yaklaşımında Geri Alma İşlemi Akış Şeması

Büyük sistemlerde her bir işlem yazılımlar tarafından yapılır, ancak bazı durumlarda insan müdahalesi gerekmektedir. Eğer ki, bir geri alma senaryosu başladığında ve bu geri alma senaryosu işlenirken bir hata oluşur ise, sistem ya ikinci bir geri alma senaryosu işletmelidir ki bunu planlayıp programlamak oldukça zordur, çünkü hangi geri alma adımında hangi hata oluşacağını bilmek çok zor olabilir. Böyle bir durumda insan müdahalesi ile gerekli düzeltmeler veri tabanında yapmalı ve gerekli servislere gerekli müdahalede bulunulmalıdır.

MSM'de geri alma senaryoları çok önemlidir. Çünkü servisler birbirinden bağımsız olduğu için geri alınma yapılacak ise, servisler birbirleri ile koordineli bir biçimde geri alma yapılmalıdır. Yönetici servis çatısı altında kurulmuş bir sistemde, geri alma senaryoları, yönetici servisi olmadan yapılan sistemlere göre daha kolay olabilmektedir. Çünkü, her bir işlemin sırası yönetici servis içerisinde belirlendiği gibi, bir hata durumunda geri alma işlemi de aynı yönetici servis üzerinden yürütüleceği için, kontrol edilmesi ve tasarlanması daha kolaydır.

2.7 Güvenlik

Yazılım projelerinde en önemli noktalardan biri de sistemin güvenliğidir. Kötü niyetli kişiler yazılımın her bir adımında bir şekilde sisteme giriş yapmaya çalışıp verileri çalma ya da verileri bozma niyetinde olabilir. Dolayısıyla güvenlik, sistemin hızlı

çalışmasından, erişilebilirliğinden, hata toleransından daha önemli olabilir. Mikroservis mimarisinde servisler birbirinden bağımsız olduğu için ve kuyruk yapısı ile ya da REST mimari ile birbirleri ile haberleştikleri için güvenlik açığı oluşturabilecek birçok nokta olabilir (Sun, Nanda, ve Jaeger, 2015). Bu nedenle mikroservislerin birbirleri ile haberleştiği kuyruk sistemlerinde kuyruk yapısının güvenlik açığı içermeyecek şekilde tasarlanması gerekmektedir. Bunun için internete kapalı sadece belli bir özel ağ sistemi içinde çalışır hale getirilebilir.

Günümüzde yazılımları servis haline getirip bunları bir düzen içinde gerekli sistemlere kurulum yapan ekipler ayrı bir departman altında toplanmaktadır. Bu departmana DevOps denmektedir. DevOps ekipleri sistemde çalışan birçok servisi uçtan uca kurulumunu ayrı bir yazılım ile otomatik bir şekilde yapılmasını sağlar. CI/CD sistemleri ile sürekli entegrasyon ve sürekli kurulum sistemlerini yönetir. Sürekli entegrasyon terimi, bir servisin kodlarını sistemde düzenli bir şekilde geliştirme, test ve canlı sistem bacakları altında güvenli bir şekilde tutmaktır. Sürekli kurulum ise, otomatik bir şekilde kod deposundan ilgili ortama kurulumun yapılmasıdır (AWS DevOps İnternet Sitesi, 2022). DevOps'un sisteme katkıları şunlardır;

Sistemin güvenilirliğini artırır.

Kurulumların hızlı yapılmasını sağlar.

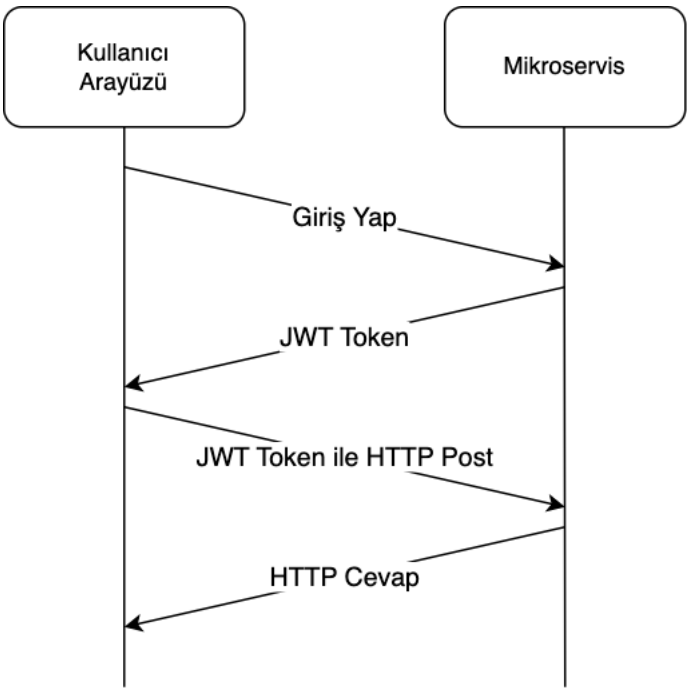
Gerektiğinde sistemin ölçeklenebilirliğini kolaylaştırır.

Sistemin güvenliğini artırır.

2.7.1 Kimlik doğrulama ve yetkilendirme

Günümüzdeki web sitelerinde, web projelerinde genellikle kullanıcı yönetimi kullanılmaktadır. Yani bir kullanıcı sisteme giriş yapar, kendi bilgileri ile ilgili işlemleri yönetir ve çıkar. Bu işlemler sırasında kullanıcı bir kullanıcı adı ve şifre ile sisteme giriş yapar. Bu adımda kullanıcı şifresi ile işlem yapıldığı için güvenlik açığı ortaya çıkabilmektedir. Bu güvenlik açığını en aza indirmek için, kullanıcı, sisteme ilk giriş yaptığında sistem ona bir web jetonu verir ve kullanıcı bu web jetonu ile işlemlerine devam eder. Günümüzde en çok kullanılan web jetonlarından biri de JWT (JSON Web

Token) olmaktadır (auth0 İnternet Sitesi, 2022). Şekil 16'da JWT çalışma prensibi görülmektedir.



Şekil 16. JWT Çalışma Prensibi

Şekil 16'da görüldüğü gibi, öncelikle kullanıcı arayüzü ilgili mikroservise kullanıcı adı ve şifresi ile bir HTTP metodu giriş yapma isteği gönderir. Başarılı giriş yapıldıktan sonra mikroservis kullanıcıya bir JWT jetonu gönderir. Artık kullanıcı arayüzü bu mikroservise istek atacağı zaman bu JWT jetonunu kullanacaktır. Daha sonra kullanıcı arayüzü mikroservise bir bu JWT jetonu ile HTTP Post isteği atar ve servis de ona ilgili cevabı döner.

Kullanıcı işlemlerinde genellikle iki aşama vardır. Birincisi kimlik doğrulama, ikincisi de kimlik yetkilendirmedir. Kimlik doğrulama işlemi sisteme giriş yapma yetkisine sahip herhangi bir kullanıcıyı teyit etme işlemine denmektedir. Kimlik yetkilendirme ise, giriş yapan kullanıcının hangi yetkilere sahip olacağını belirler. Böylelikle giriş yapan kullanıcının, sıradan bir kullanıcı, yetkili kullanıcı ya da süper kullanıcı gibi sistemde farklı işlemler yapabilecek kullanıcılar olup olmadığını belirler (auth0 İnternet Sitesi, 2022).

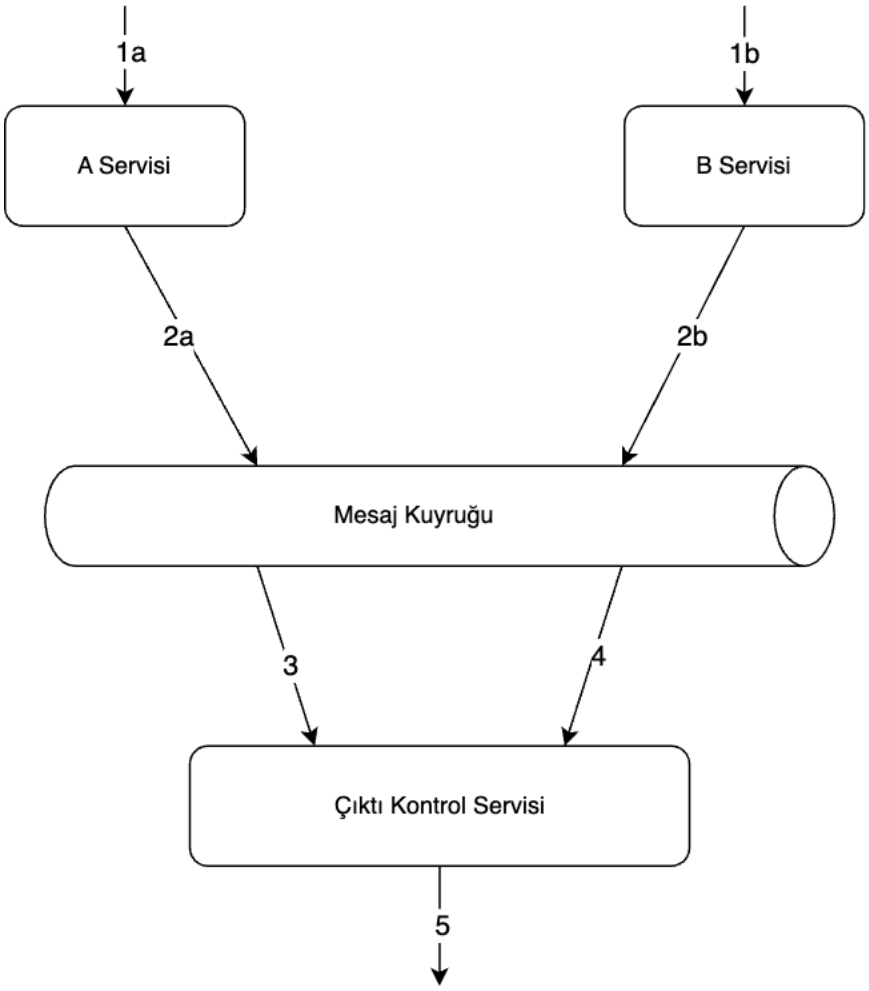
3. MİKROSERVİS MİMARİSİNDE DAĞITIK İŞLEMLERİN YÖNETİMİNE YENİ ÇÖZÜMLER

Dağıtık işlemlerin çözümlerinde sistemin asenkron veya senkron olması ve kuyruk yapısı kullanılıp kullanılmamasına göre farklı yöntemler seçilebilmektedir. 2PC ve Saga yönteminin kullanılması ve bunlara uygun geri alma senaryoları uygulanması MSM'de dağıtık işlemlerin yönetimini kolaylaştırmaktadır. Burada karşılaşılan zorluklardan biri, geri alma senaryoların işlenmesi olmaktadır. Çok fazla mikroservisin kullanıldığı sistemlerde geri alma işlemlerini tasarlamak ve yönetmek zorlaşmaktadır. Bu zorluklara alternatif bir çözüm olarak tüm mikroservislerin çıktılarını dinleyen ve geri alma işlemlerini de kendisi yapabilen çıktı kontrol servisi isimli ile yeni bir servis ele alınacaktır.

Çıktı kontrol servisi orkestrasyon ya da koreografi yaklaşımı ile oluşturulmuş Saga mimarisinde çalışan tüm mikroservislerin çıktılarına abone olmuş bir halde tüm servisleri dinler ve bir işlemin başarılı ya da başarısız bittiğine karar vererek kullanıcıyı bilgilendirir. Aynı zamanda olası bir hata durumunda tüm geri alma işlemini kendisi yaparak asıl iş yapan iç mikroservislere geri alma işlemi gibi ek bir yük binmesine engel olmaktadır. Böylelikle sistem daha stabil ve hızlı çalışabilir.

Geri alma işlemlerini ayrıca takip eden ve işleyen ayrı bir servisin varlığı altında, bağımsız mikroservisler aynı anda işleme başlayabilmektedir. Böylece asenkron işlemler ile sistemin çok daha hızlı çalışması sağlanmaktadır. Çıktı kontrol servisi, mikroservislerden birinden hata mesajı yakalar ise o işleme tabi olan daha önce çalışmış servislerin yaptığı işlemi kendisi geri alır. Burada dikkat edilmesi gereken husus, bir işlemin farklı birçok mikroservis ile çalıştığı durumlarda bir işlem kimliği ile her bir mikroserviste yürütülen işlemin hangi ana işleme ait olduğunu takip etmektir. Böylelikle çıktı kontrol servisinin aldığı bir hata mesajı ile buradaki işlem kimliğine sahip olan diğer servislerin yaptığı işlemi geri alır. Hata alınmaz ve tüm işlemler başarılı biter ise, çıktı kontrol servisi bu işlem kimliğini takip ederek tüm mikroservislerden başarılı sonuçlarını birleştirir ve kullanıcıya ya da kuyruğa işlemin başarılı bittiği mesajını verir.

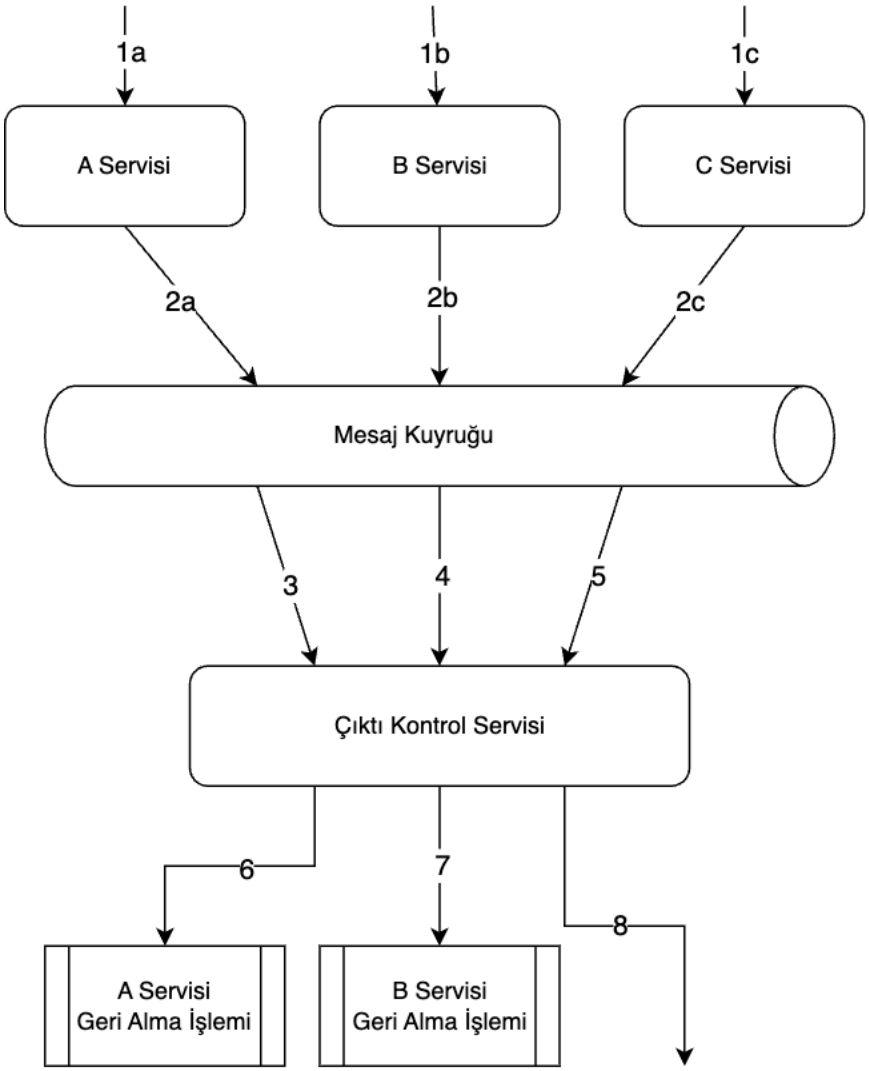
Şekil 17’de çıktı kontrol servisi ile kontrol edilen koreografi Saga mimarisi ile hazırlanmış bir sistem yer almaktadır. Bu senaryoda, sistemde var olan iki mikroservis de başarılı sonuç üretmiştir



Şekil 17. Başarılı Çalışan Mikroservisleri Kontrol Eden Çıktı Kontrol Servisi

Şekil 17’de görüldüğü gibi, mikroservisler paralel tetiklenmektedir. 1a ve 1b adımlarında A ve B servisi tetiklenir. Yaklaşık aynı zamanda işlemlerini bitiren servisler 2a ve 2b adımlarında başarılı bitirdikleri sonucunu mesaj kuyruğuna gönderir. Mesaj kuyruğundan 3. ve 4. çıktı kontrol servisi bu iki mesajı mesajda var olan işlem kimliği ile eşleştirip iki işlemin aynı ana işleme ait olduğu kararına varır ve bu iki mesaj da başarılı biten bir sonuç olduğu için 5. adımda ana işlemin başarılı bittiğini sonuç olarak döner.

Şekil 18'te ise çalışan bir mikroservisin hata alması durumunda çıktı kontrol servisin alacağı aksiyon görselleştirilmiştir. 1a, 1b ve 1c adımlarında A, B ve C servisleri paralel çalıştırılmıştır. Ardından bu servisler işlemlerini 2a, 2b ve 2c adımlarında bitirip kuyruğa bu işlemlerinin sonucunu yazmışlardır. Ancak C servisi işlemini başarısız bir şekilde bitirmiştir. Bu durumda çıktı kontrol servisi 3. 4. ve 5. adımlarda bu 3 mesajı alır. Mesajların içerisinde var olan işlem kimliği ile bu 3 mesajın aynı ana işleme ait olduğunu anlar ve C servisinin başarısız bitirmesinden dolayı geri alma işlemini başlatır. Geri alma işlemleri 6. ve 7. adımlarda A ve B servislerinin yaptıkları işlemleri geri alma olarak uygulanır. Bu 6. ve 7. adımlarda geri alma işlemini A ve B servisleri değil, bizzat çıktı kontrol servisi yapar. 8. adımda ise ana işlemin başarısız bittiğini, ama geri alma işleminin başarılı bittiğini sonuç olarak döner.

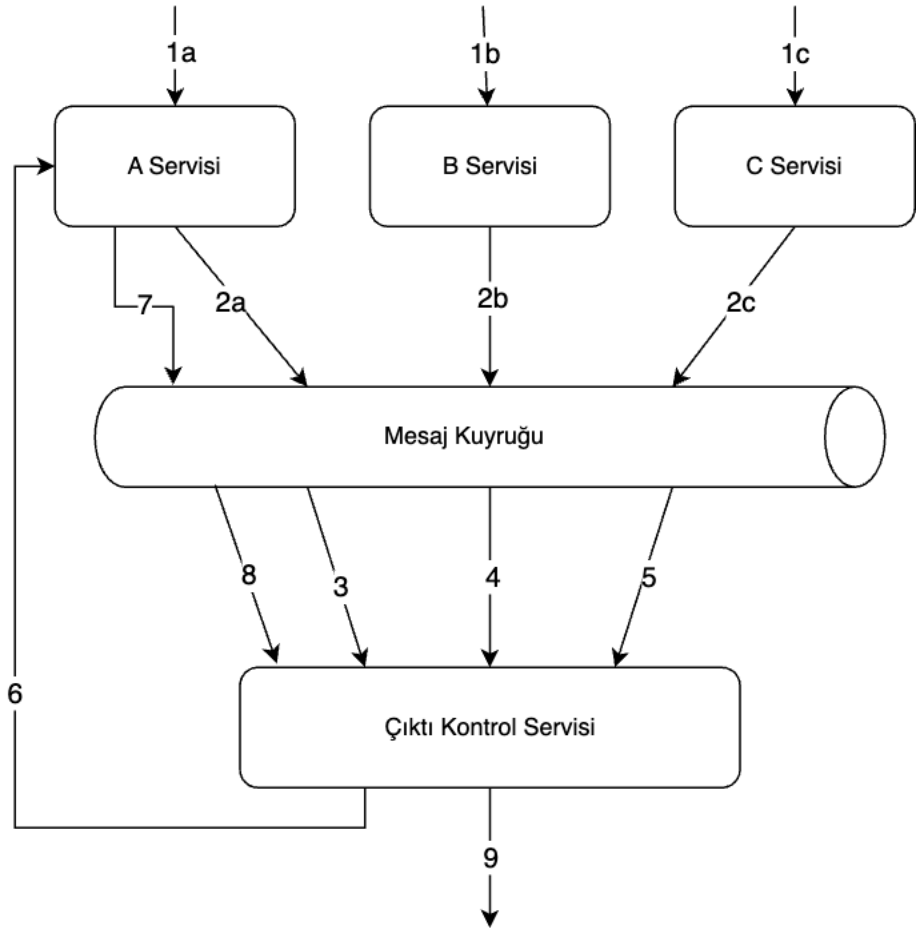


Şekil 18. Geri Alma İşlemi Yapan Çıktı Kontrol Servisi

Çıktı kontrol servisi geri alma işlemlerinin yanı sıra, hata alınması durumunda geri alma işleminden önce tekrar deneme işlemini de yapabilir. Belli bir ayarlama ile kaç kere ve ne kadar süre aralığı içerisinde tekrar denemesi kararlaştırılır. Bu tekrar denemelerde başarılı olunur ise sistem ana işlemi başarılı bitirir. Tekrar denemelerde de başarısız olunur ise çıktı kontrol servisi geri alma işlemlerini yapar ve ana işlemin başarısız bittiğine dair bilgiyi çıktı olarak verir.

Şekil 19'da tekrar deneme mekanizması ile çalışan çıktı kontrol servisi uygulaması görülmektedir. Bu senaryoda, 1a, 1b ve 1c adımlarında A, B ve C servisleri çağırılır. Aynı şekilde 2a, 2b ve 2c adımlarında servisler işlemlerini bitirip mesaj kuyruğuna sonuçlarını yazar. Ancak A servisi sonucunu başarısız bitirmiştir. Bu durumda çıktı kontrol servisi 3.

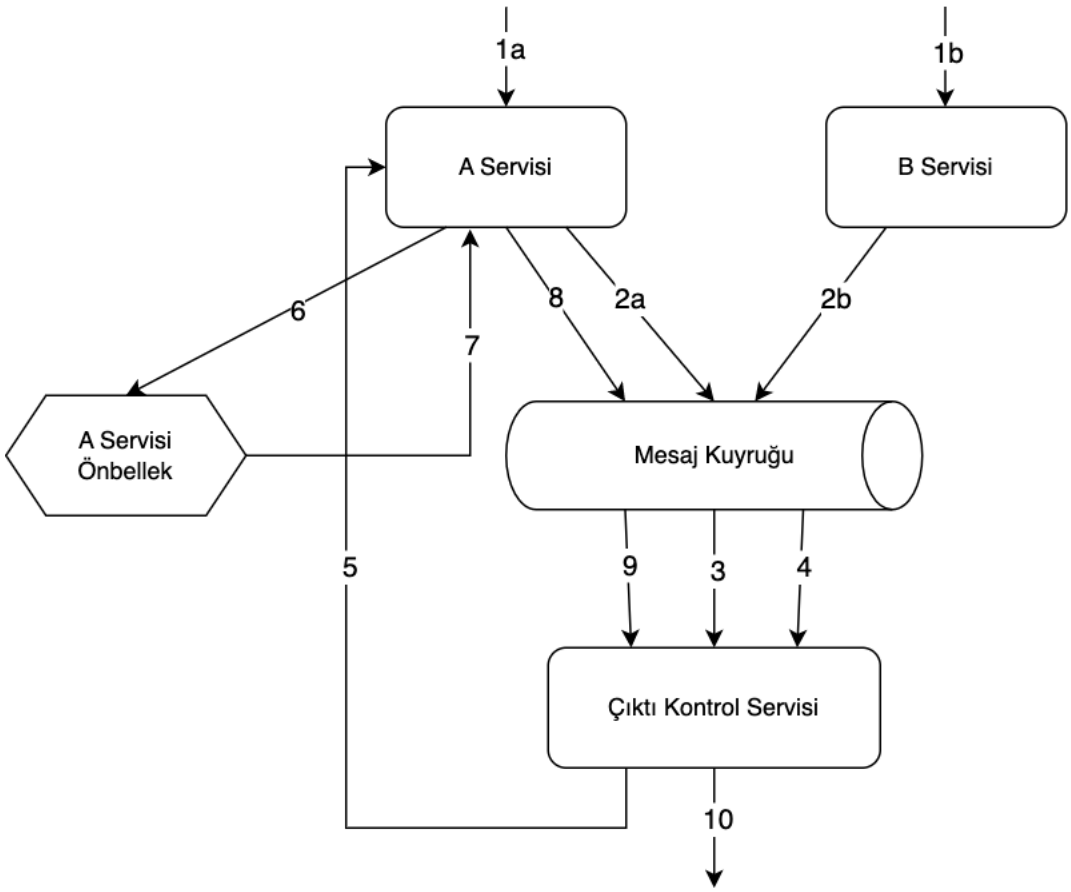
4. ve 5. adımlarda bu mesajları alır ve A servisinin yaptığı işlemi geri alma adımından önce 6. adımda tekrar deneme işlemi uygular. Bu tekrar deneme işlemi ile başarılı olur ve A servisi 7. adımda başarılı bittiğine dair mesajı kuyruğa gönderir. 8. adımda kuyruktan mesajı alan çıktı kontrol servisi artık tüm işlemler başarılı bittiği için 9. adımda tüm işlemin başarılı bittiğini çıktı olarak verir.



Şekil 19. Tekrar Deneme İşlemi Yapan Çıktı Kontrol Servisi

Tekrar deneme işlemi uygulanacak ise dikkat edilmesi gereken nokta, tekrar denenecek isteğin bir yerde muhakkak tutulması olacaktır. Sanki istek tekrar ilgili servise gönderiliyor gibi tekrar atılmalıdır. Şekil 19'deki örnekte servisler hata alması durumunda işlem için gönderilen girdi isteğini kuyruktaki mesajına koyar ve çıktı kontrol servisi bu girdi isteğini tekrar deneme işleminde girdi olarak kullanır. Burada çıktı kontrol servisi geri alma işlemindeki gibi tekrar deneme işlemini kendisi yapmaz, ilgili servisi

tekrar çağırır. Buradaki handikap, servisi tekrar çalıştırmak için servisin tetiklendiği sıradaki girdinin aynı şekilde tutulup kuyruğa gönderilmesi olacaktır. Eğer burada şifre gibi hassas bilgi var ise bunu kuyruğa atmak bazı güvenlik sorunları oluşturabilir. Bunun önüne geçmek için, her bir servis tekrar denenme isteğine tedbir olarak çalıştırıldığı girdiyi bir geçici bellekte tutarak bunun tekrar çağırılması sağlanabilir. Böylelikle kuyruğa detaylı girdi bilgisi gönderilmemiş olur. Şekil 20'de önbellekte tutulan servis girdilerinin tekrar tetikleme işleminde kullanıldığı bir mimari görselleştirilmiştir.



Şekil 20. Önbellek ile Tekrar Deneme İşlemi Yapan Çıktı Kontrol Servisi

Şekil 20'de görüldüğü gibi, A ve B servisleri 1a ve 1b adımlarında kullanıcı tarafından tetiklenir. 2a ve 2b adımlarında A ve B servisleri çıktılarını kuyruğa gönderir. Çıktı kontrol servisi 3. ve 4. adımlarda bu mesajları alıp işlemeye başlar. A servisinin çıktısında servisin başarısız bittiğini anlayıp 5. adımda A servisine aynı işlemi tekrarlaması için

komut gönderir. A servisi önbellekte hangi girdinin tekrarlanacağını çıktı kontrol servisinin kuyruktan aldığı mesajın benzersiz tanımlayıcı kimliği ile anlamaktadır. 6. adımda A servisi önbellekten bu benzersiz tanımlayıcı kimliğe bağlı girdi kaydını alma isteği gönderir ve 7. adımda alır ve işlemi tekrarlar. A servisi işlemin başarılı bittiği mesajını 8. adımda mesaj kuyruğuna gönderir. 9. adımda çıktı kontrol servisi bu başarılı mesaj çıktısını alır ve hem A hem de B servisinin başarılı bittiği için 10. adımda tüm işlemin başarılı bittiği bilgisini çıktı olarak döner.

4. BULGULAR VE TARTIŞMA

Saga yöntemi ile geliştirilen bir yazılım projesinde orkestrasyon veya koreografi yaklaşımları kıyaslanacak olunursa, her iki yöntemin de avantaj ve dezavantajları olabilmektedir. Sadece sistemin sağlıklı çalışabilmesi gibi sistemsel olaylar değil, yazılımı geliştirecek ekibin yetkinliği, ekibin büyüklüğü gibi değişkenler de seçilecek olan yaklaşımı etkileyebilir. Aynı şekilde çıktı kontrol servisinin avantaj ve dezavantajları da bulunmaktadır. Bu yaklaşımların karşılaştırmaları bu başlık altında incelenecektir.

4.1 Bulgular

Tablo 1'de Saga yönteminde var olan orkestrasyon ve koreografi yaklaşımlarının karşılaştırılması görülmektedir. Çalışma yöntemi başlığı, sistemin aynı anda çoklu işlem yapabilme kapasitesi olarak düşünülebilir. MM tek bir uygulama içerisinde çalışan servisler bütünü olduğu için sistem bir tanedir ve tekil çalışma yöntemi vardır. 2PC'de ise sistemin uygunluğuna göre hem tekil hem de çoğul bir şekilde sistem tasarlanabilir. Saga yönteminde ise ana işlem birçok mikroservise bölündüğü için çoklu çalışma yöntemi vardır. Senkron ya da asenkron çalışma olarak incelendiğinde ise MM ve 2PC yöntemiyle oluşturulan projelerde, sistem senkron olarak çalışmaktadır. Yani bir işlem bitmeden diğer bir işleme başlanmaz. Aynı anda sadece tek bir işlem çalışır. Saga yönteminde ise orkestrasyon yaklaşımında hem senkron hem asenkron çalıştırılabilir. Yönetici servis bu yöntemi ayarlayabilir. Koreografi yaklaşımında ise sistem, arkada tamamen bağımsız yönetici servisi olmayan mikroservisler olduğu için asenkron çalışma prensibine sahiptir. Kontrol noktası olarak, yönetici bir servisi olmayan Saga koreografi yaklaşımı dağıtık olarak çalışırken diğerleri merkezi çalışmaktadır. ACID prensiplerine uyum olarak incelendiğinde ise, Saga yöntemi ACID prensiplerinden sadece izolasyon terimini ihlal ederken MM ve 2PC ile oluşturulan sistemlerde ACID prensipleri tamamen karşılanabilir (Ibryam, 2021).

Tablo 1. Yazılım Mimarileri Özellik Karşılaştırması

	Monolitik Mimari	**2PC**	**Saga Orkestrasyon Yaklaşımı**	**Saga Koreografi Yaklaşımı**
Çalışma Yöntemi	Tekil	Tekil/Çoğul	Çoğul	Çoğul
Senkron/Asenkron	Senkron	Senkron	Senkron/Asenkron	Asenkron
Kontrol Noktası	Merkezi	Merkezi	Merkezi	Dağıtık
ACID Prensip Uyumu	ACID	ACID	ACD	ACD

4.2 Tartışma

Çıktı kontrol servisi, Saga yönteminin her iki yaklaşımı ile de uygulanabilir. Ama orkestrasyon yaklaşımında zaten var olan bir yönetici servisin olması, ikinci bir yönetici servisin eklenmesi ile bir kısır döngüye girebilir. Bir sistemde iki yönetici servisin olması, işlemlerin yanlış akmasına sebep olabilir. Bu gibi durumlarda istekleri kullanıcıdan alan yönetici servisin sadece istekleri doğru bir şekilde iletmesi ile sorumlu olup, çıktı kontrol servisinin ise sadece çıktıları kontrol eden bir servis olarak kalması gerekmektedir.

Çıktı kontrol servisinin sisteme kattığı önemli kazanımlar, özellikle koreografi yaklaşımındaki mikroservisler arasındaki iletişim karmaşıklığını azaltması ve geri alma senaryosunu kolaylaştırması olarak sayılabilir. Buna ek olarak, sistemin paralel çalışmasını sağladığı için, sistemin çok daha hızlı çalışmasını sağlamış olur. Aynı zamanda bir yönetici servis altında senkron çalışan mikroservislerin olduğu bir mimariye de entegre edilebilir.

Her mimarinin olduğu gibi çıktı kontrol servisinin de dezavantajları bulunmaktadır. Bunlar, birbirleri ile bağımlı çalışan mikroservisler var ise, geri alma adımları da birbirleri ile bağımlı olacaktır ve tek bir noktada geri alma işlemi yapıldığı için karmaşıklık doğabilir. Bunun yanı sıra, yönetici servisin dezavantajlarından biri olan tekil yönetim noktası da bir dezavantaj sayılabilir. Çıktıları tek bir yerde kontrol etmek, eğer ki çıktı

kontrol servisi hata alır ve çalışamaz duruma gelir ise sistem de aynı şekilde tamamen cevap veremez hale gelebilir.

Çıktı kontrol servisinin bir diğer dezavantajı ise, geri alma senaryolarının kendisi işletmesi ve çalıştırması olabilir. Her servisin kendi geri alma işlemini yapması servislere fazladan iş yükü bindirilmesine neden olur, ancak her servis kendi işlemini kontrol altında tutar. Çıktı kontrol servisi bu geri alma işlemini servislerden alması ile çalışan mikroservislerden geri alma işlemi yükünü almasına karşın, bu iş yükünü kendisine bindirir. Çok fazla mikroservisin var olduğu sistemlerde bu geri alma işlemlerini tek tek çıktı kontrol servisine eklemek ve bunları yönetmek zor olabilir.

5. SONUÇ ve ÖNERİLER

Büyük yazılım projelerinde bir sistem tasarlanırken hangi yöntemin, hangi yaklaşımın seçileceği, öncelikle projenin nasıl bir iş yapısına sahip olduğu göz önüne alınmalı ve bu iş yapısına uygun olan mimarinin bu projeye uygunluğu incelenmelidir. Aynı zamanda bu yazılımı geliştirecek olan ekibin büyüklüğü, yetkinliği de önemlidir. Çok büyük sistemler tasarlanıp geliştirilirken, birçok kişinin çalıştığı birçok farklı ekip görev almaktadır. Örneğin Saga yönteminin koreografi yaklaşımı ile çalışan bir sistemde yüzlerce mikroservis var ise bunları yönetmek, birbirleri ile bağlantılarını izlemek çok zor olabilir. Bu gibi durumlarda çıktı kontrol servisi tüm bu servislerin çıktılarını dinleyerek, tekil bir noktada toplayıp gerekli işlemin yapılmasını sağlayabilir. Ancak burada sistemin sağlıklı çalışması için mikroservislerin birbirleri ile tamamen bağımsız olması yani birinin çıktısı başka birinin girdisi olmayacak şekilde ya da en az olacak şekilde tasarlanmalıdır. Saga yönetminin orkestrasyon yaklaşımına da çıktı kontrol servisi rahatlıkla eklenebilir. Orkestrasyon yaklaşımında yönetici servis kullanıcıdan gelen veriyi ilgili mikroservislere yönlendirir. Ayrıca ilgili işlemin gerçekleştirilmesi için bir düzen içinde farklı mikroservisleri çağırabilir. Çıktı kontrol servisi ise yönetici servisin yaptığı işlere karışmadan, sadece çıktılara odaklanarak geri alma ve tekrar deneme işlemlerini yapar. Dolayısıyla iki yönetici servisin birbirleri ile çakışma durumu olmaz.

Sistemdeki alt mikroservislerden geri alma yükünü çıktı kontrol servisinin alması, mikroservislerin asıl amacı olan gerekli işlemi yapmasına odaklanmasını sağlar. Normal ve stabil bir sistemde çalışan bir mikroserviste bir hata oluşma ihtimali olabildiğince düşük olmalıdır. Bu nedenle geri alma işlemi de düşük ihtimalle gerçekleşmektedir. Dolayısıyla nadiren çalışacak olan geri alma işlemi ile ilgili kısımları mikroservisten alıp çıktı kontrol servisine vermek, genel olarak sistemin daha stabil ve işine odaklı gereksiz kodlar taşımadan çalışmasını sağlar.

Çıktı kontrol servisi, burada bahsedilen ve bahsedilmeyen tüm mimarilere eklenebilir. Sadece çıktıları kontrol edip, geri alma senaryolarını ve tekrar deneme mekanizmasını işlediği için her bir mimariye sonradan eklenebilir. Yani hali hazırda çalışan, geliştirmesi tamamlanmış bir sisteme çok az bir efor ile entegre edilebilir. Bu ekleme yapıldıktan

sonra, eğer sistem asenkron çalışıyor ise, sistem paralel çalışır hale getirilip daha verimli bir hale getirilebilir. Aynı şekilde uygun bir önbellek yönetimi ile tekrar deneme işlemlerini de çıktı kontrol servisi ele alabilir. Böylelikle sistemdeki mikroservisler sadece yapması gereken gerekli işlemi yapar.

6. KAYNAKÇA

Aksakalli, I. K., Çelik, T., Can, A. B., ve Tekinerdoğan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. *Journal of Systems and Software.*

Al-Debagy, O., ve Martinek, P. (2018). A comparative review of microservices and monolithic architectures. *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 149-154.

Al-Houmaily, Y. (2010). Atomic commit protocols, their integration, and their optimizations in distributed database systems. *International Journal of Intelligent Information and Database Systems.*

Al-Houmaily, Y. J., ve Samaras, G. (2009). Two-Phase Commit. *Encyclopedia of Database Systems* (s. 3204-3209). içinde Springer.

auth0 İnternet Sitesi. (2022). *Introduction to JSON Web Tokens*. Erişim Tarihi: 09.11.2022 https://jwt.io/introduction adresinden alınmıştır.

AWS ECR İnternet Sitesi. (2022). *Amazon Elastic Container Registry*. Erişim Tarihi: 02.11.2022 https://aws.amazon.com/tr/ecr/ adresinden alınmıştır.

AWS Fargate İnternet Sitesi. (2022). *Fargate*. Erişim Tarihi: 02.11.2022 https://aws.amazon.com/tr/fargate/ adresinden alınmıştır.

AWS DevOps İnternet Sitesi. (2022). *DevOps Model Defined*. Erişim Tarihi: 09.10.2022 https://aws.amazon.com/devops/what-is-devops adresinden alınmıştır.

Cebeci, K., ve Koçak, Ö. (2020). Design of an Enterprise Level Architecture Based on Microservices. *Bilgi Teknolojileri Dergisi, 13*(4), 357-371.

Costa, B., Pires, P. F., Delicato, F. C., ve Merson, P. (2014). Evaluating a Representational State Transfer (REST) Architecture. *IEEE/IFIP Conference on Software Architecture*, (s. 105-114).

Dürr, K., Lichtenthaler, R., ve Wirtz, G. (2021). *An Evaluation of Saga Pattern Implementation Technologies.* Bamberg, Germany: 13th Central European Workshop on Services and their Composition.

Daraghmi, E., Zhang, C.-P., ve Yuan, S.-M. (2022). Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. *Applied Sciences, 12*(12), 6242.

Dobbelaere, P., ve Esmaili, K. S. (2017). Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. *The 11th ACM International Conference.*

Docker İnternet Sitesi. (2022a). *Use containers to Build, Share and Run your applications*. Erişim Tarihi: 30.10.2022 https://www.docker.com/resources/what-container/ adresinden alınmıştır.

Docker İnternet Sitesi. (2022b). *Get Started*. Erişim Tarihi: 01.11.2022 https://docs.docker.com/get-started/ adresinden alınmıştır.

Docker İnternet Sitesi. (2022c). *Docker Compose Overview*. Erişim Tarihi: 01.11.2022 https://docs.docker.com/compose/ adresinden alınmıştır.

Docker İnternet Sitesi. (2022d). *Docker Hub*. Erişim Tarihi: 02.11.2022 https://www.docker.com/products/docker-hub/ adresinden alınmıştır.

Docker İnternet Sitesi. (2022e). *Docker Desktop*. Erişim Tarihi: 02.11.2022 https://www.docker.com/products/docker-desktop/ adresinden alınmıştır.

Docker İnternet Sitesi. (2022f). *Docker Machine Overview*. Erişim Tarihi: 02.11.2022 https://docker-docs.netlify.app/machine/overview/ adresinden alınmıştır.

Fielding, T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* University of California, Irvine.

Garcia-Molina, H., ve Salem, K. (1987). Sagas. *ACM SIGMOD Record, 16*(3), 249-259.

Golub, M., Jakobović, D., ve Janeš, I. (2005). An overview of distributed programming techniques. *Hypermedia and Grid Systems, MIPRO* (s. 215-220). Opatija, Croatia: Sun Microsystems.

gRPC İnternet Sitesi. (2022a). *gRPC*. Erişim Tarihi: 30.10.2022 https://grpc.io/ adresinden alınmıştır.

gRPC İnternet Sitesi. (2022b). *Introduction to gRPC*. Erişim Tarihi: 08.11.2022 https://grpc.io/docs/what-is-grpc/introduction/ adresinden alınmıştır.

Protocol Buffers İnternet Sitesi. (2022). *Protocol Buffers*. Erişim Tarihi: 08.11.2022 https://developers.google.com/protocol-buffers/ adresinden alınmıştır.

Hohpe, G. (2005). Your coffee shop doesn't use two-phase commit [asynchronous messaging architecture]. *IEEE Software, 22*(2), 64-66.

Ibryam, B. (2021). *Distributed transaction patterns for microservices compared*. 10 30, 2022 tarihinde https://developers.redhat.com/articles/2021/09/21/distributed-transaction-patterns-microservices-compared#how_to_choose_a_distributed_transactions_strategy adresinden alındı

Kalske, M. (2018). Transforming monolithic architecture towards microservice architecture. Helsinki: Helsingin yliopisto.

Kocaman, Y. (2018). *Mikroservis Tabanlı Ödeme Sistemi Tasarımı ve Gerçeklenmesi.* İstanbul: Okan Üniversitesi.

Kratzke, N. (2015). About microservices, containers and their underestimated impact on network performance. *Cloud Computing 2015 at Nice.* Nice.

Kreps, J., Narkhede, N., ve Rao, J. (2011). Kafka: a Distributed Messaging System for Log Processing. *Proceedings of the NetDB, 11*, 1-7.

Kubernetes İnternet Sitesi. (2022a). *Kubernetes*. Erişim Tarihi: 30.10.2022 https://kubernetes.io/ adresinden alınmıştır.

Kubernetes İnternet Sitesi. (2022b). *Kubernetes Overview*. Erişim Tarihi: 02.11.2022 https://kubernetes.io/docs/concepts/overview/ adresinden alınmıştır.

Li, L., Ye, C., ve Zhou, H. (2022). Cache Replacement Algorithm Based on Dynamic Constraints in Microservice Platform. *2022 International Conference on Service Science (ICSS)* (s. 167-174). içinde

Medjahed, B., Ouzzani, M., ve Elmagarmid, A. (2009). Generalization of ACID Properties. *Encyclopedia of Database Systems* (s. 1221-1222). içinde Springer.

Microservies.io İnternet Sitesi. (2022). *Pattern: Microservice Architecture*. Erişim Tarihi: 30.10.2022 https://microservices.io/patterns/microservices.html adresinden alınmıştır.

Microsoft Azure İnternet Sitesi. (2022). *Azure Container Registry*. Erişim Tarihi: 02.11.2022 https://azure.microsoft.com/en-us/products/container-registry/ adresinden alınmıştır.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems.* O'ReillyMedia Inc.

Nielsen, D. (2015). Investigate Availability and Maintainability within a Microservice Architecture. Aarhus: Department of Computer Science, Aarhus University.

Redis İnternet Sitesi. (2022). *Redis*. Erişim Tarihi: 05.11.2022 https://redis.io/ adresinden alınmıştır.

Rudrabhatla, C. K. (2018). Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture. *(IJACSA) International Journal of Advanced Computer Science and Applications, 9*(8).

Samaras, G., Britton, K., Citron, A., ve Mohan, C. (1993). Two-phase commit optimizations and tradeoffs in the commercial environment. *IEEE 9th International Conference on Data Engineering.*

Saransig, A., ve Tapia Leon, F. (2019). Performance Analysis of Monolithic and Micro Service Architectures – Containers Technology: Proceedings of the 7th International Conference on Software Process Improvement (CIMPS 2018). *Trends and Applications in Software Engineering* (s. 270-279). içinde

Serrano, N., Hernantes, J., ve Gallardo, G. (2014). Service-Oriented Architecture and Legacy Systems. *Software, IEEE, 31*, 15-19.

Sharvari, T., ve Sowmya Nag, K. (2019). *A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming.* CoRR abs/1912.03715.

Stefanko, M., Chaloupka, O., ve Rossi, B. (2019). The Saga Pattern in a Reactive Microservices Environment. *In Proceedings of the 14th International Conference on Software Technologies.* Prague.

Sun, Y., Nanda, S., ve Jaeger, T. (2015). Security-as-a-Service for Microservices-Based Cloud Applications. *IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom).* Vancouver.

Tapia, F., Mora, M., Fuertes, W., Aules, H., Flores, E., ve Toulkeridis, T. (2020). From Monolithic Systems to Microservices: A Comparative Study of Performance. *Applied Sciences, 10*, 5797.

Villamizar, M., Garcés, O., Castro, H., Verano Merino, M., Salamanca, L., Casallas, R., ve Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. *2015 10th Computing Colombian Conference (10CCC).* Bogota.

Weerasinghe, L. D., ve Perera, I. (2021). An exploratory evaluation of replacing ESB with microservices in service-oriented architecture. *International Research Conference on Smart Computing and Systems Engineering (SCSE).*

Yang, D., Gao, Y., He, W., ve Li, K. (2021). Design and Achievement of Security Mechanism of API Gateway Platform Based on Microservice Architecture. *Journal of Physics: Conference Series, 1738*(1).

Zhong, L., Zheng, X., Liu, Y., Wang, M., ve Cao, Y. (2020). Cache hit ratio maximization in device-to-device communications overlaying cellular networks. *China Communications, 17*(2), 232-238.

ÖZGEÇMİŞ

Mustafa GÖRDESLİ

Eğitim

Lisans	2012	Eskişehir Osmangazi Üniversitesi, Fen Bilimleri Enstitüsü, Elektrik Elektronik Mühendisliği Bölümü

İş/İstihdam

2021	Kıdemli Yazılım Geliştirme Mühendisi