T.C.
SOSYAL GÜVENLİK KURUMU BAŞKANLIĞI

MİKROSERVİS MİMARİSİNİN İNCELENMESİ VE MONOLİTİK BİR SGK YAZILIM UYGULAMASININ MİKROSERVİS MİMARİSİNE DÖNÜŞTÜRÜLMESİ SÜRECİNİN İLK AŞAMASINA DAİR ÖRNEK ÇALIŞMA

Sosyal Güvenlik Uzmanlığı Tezi

Hazırlayan
Hallaç Mansur AKBAŞ

Tez Danışmanı
Murat BİNAY
Sosyal Güvenlik Uzmanı

Ağustos 2024
Ankara

2024

MİKROSERVİS MİMARİSİNİN İNCELENMESİ VE MONOLİTİK BİR SGK YAZILIM UYGULAMASININ MİKROSERVİS MİMARİSİNE DÖNÜŞTÜRÜLMESİ SÜRECİNİN İLK AŞAMASINA DAİR ÖRNEK ÇALIŞMA **Hallaç Mansur AKBAŞ**

Hallaç Mansur AKBAŞ tarafından hazırlanan bu çalışmanın Sosyal Güvenlik Uzmanlığı Tezi olarak uygun olduğunu onaylarım. …./…./……

Tez Danışmanının,

Adı SOYADI: Murat BİNAY

İmzası:

Bu çalışma, Kurulumuz tarafından oy birliği / oy çokluğu ile Sosyal Güvenlik Uzmanlığı Tezi olarak kabul edilmiştir.

	Adı Soyadı ve Unvanı	**İmzası**
Başkan	: ……	…...……..
Üye	: …….	…...……..
Üye	: ……	…...……..
Üye	: …….	…...……..
Üye	: ……	…...……..

Tarih: …./…./……

BİLİMSEL ETİK BİLDİRİM BEYANI

Sosyal Güvenlik Kurumu Sosyal Güvenlik Uzmanlık Tezi ve Yeterlik Sınavı Yönergesine uygun olarak hazırladığım bu tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, ayrıca tez yazım kurallarına uygun olarak hazırlanan bu çalışmada bana ait olmayan her türlü ifade ve bilginin kaynağına eksiksiz atıf yapıldığını beyan eder, tezimde yer alan ve sahiplerinden yazılı izin alınarak kullanılması zorunlu olan metinleri yazılı izin alarak kullandığımı ve istenildiğinde bu yazılı izinlerin suretlerini Kuruma teslim etmeyi taahhüt ederim. Tezimle ilgili sunduğum beyan ve taahhütlere aykırı bir durumun saptanması halinde ortaya çıkacak tüm hukuki sonuçları kabul ettiğimi beyan ederim.

…./…./……..

Hallaç Mansur AKBAŞ

YAYIMLAMA VE KULLANIM HAKLARI BEYANI

Sosyal Güvenlik Kurumu tarafından onaylanan uzmanlık tezimin tamamının veya herhangi bir kısmının Sosyal Güvenlik Kurumu tarafından basılı (kâğıt), mikroform veya elektronik formatta arşivlenmesine, kullanıma açılmasına ve Kurum nezdinde yapılacak her türlü çalışmada (analiz, rapor, kitap, süreli/süresiz yayınlar, vb.) kullanılmasına, tezle ilgili fikri mülkiyet haklarının ve tezin tamamının ya da bir bölümünün gelecekteki çalışmalarda (makale, kitap, lisans ve patent vb.) kullanım haklarının bende kalması koşuluyla izin verdiğimi bildiririm.

…./…./……..

Hallaç Mansur AKBAŞ

ÖZET

Sosyal Güvenlik Uzmanlığı Tezi

MİKROSERVİS MİMARİSİNİN İNCELENMESİ VE MONOLİTİK BİR SGK YAZILIM UYGULAMASININ MİKROSERVİS MİMARİSİNE DÖNÜŞTÜRÜLMESİ SÜRECİNİN İLK AŞAMASINA DAİR ÖRNEK ÇALIŞMA

Hallaç Mansur AKBAŞ

Bilişim teknolojileri, geçmişten günümüze sürekli ivme kazanarak değişim ve gelişim göstermiştir. Buna paralel olarak, yazılım uygulamalarının kapsamı ve işlem kapasitesi de büyümüştür. Bu büyüme, yazılımların hacminin ve karmaşıklığının artmasına yol açmıştır. Bu probleme getirilen çözümlerden biri, 2013'ten itibaren kullanımı giderek artan yeni bir yazılım mimarisi olan mikroservis mimarisidir.

Sosyal Güvenlik Kurumunda şimdiye kadar mikroservis mimarisi ile geliştirilen yazılım uygulaması olmamıştır. Bu mimarinin getirdiği çözümler ortadadır. Bu çalışmada, mikroservis mimarisinin uygulanabilmesi için kurumun hazırlık durumunun hem insan kaynağı hem teknolojik altyapı bakımından değerlendirilmesi ve monolitik mimariden mikroservis mimarisine geçiş durumunda atılması gereken adımların uygulamalı bir şekilde anlatılması amaçlanmıştır. Bunun için yazılım mimarileri ve ilgili yazılım araçları teorik bazda incelenmiş ardından seçilen örnek bir monolitik uygulama üzerinden mikroservis mimarisine geçişin ilk adımlarında neler yapılması gerektiği anlatılmıştır.

Çalışma sonucunda, monolitik bir uygulama başarılı bir şekilde ayrıştırılarak mikroservis mimarisine geçişin ilk adımı gerçekleştirilmiştir. Ayrıca personelin bilgi eksikliği ve kurumun altyapı yetersizliği tespit edilerek kurumda yapılacak bir geçiş için öngörüde bulunulabilmesi kolaylaştırılmıştır.

Anahtar Kelimeler: Mikroservis, Yazılım Mimarisi, Konteynerizasyon, Ölçeklenebilirlik

ABSTRACT

Social Security Expertise Thesis

EXAMINATION OF MICROSERVICE ARCHITECTURE AND A CASE STUDY ON THE FIRST PHASE OF TRANSFORMING A MONOLITHIC SGK SOFTWARE APPLICATION INTO A MICROSERVICE ARCHITECTURE

Hallaç Mansur AKBAŞ

Information technologies have changed and developed at an accelerated pace from the past to the present. In parallel with this, the scope and processing capacity of software applications have also grown. This growth has led to an increase in the volume and complexity of software. One of the solutions to this problem is microservice architecture, a new software architecture that has been increasingly used since 2013.

Until now, the Social Security Institution has not had any software applications developed with microservice architecture. The benefits offered by this architecture are evident. This study aims to evaluate the readiness of the organization in terms of both human resources and technological infrastructure for the implementation of microservice architecture and to explain the steps to be taken in the case of transitioning from monolithic architecture to microservice architecture practically. For this purpose, software architectures and related software tools were examined on a theoretical basis, and the initial steps of the transition to microservice architecture were explained through a selected sample monolithic application.

As a result of the study, a monolithic application was successfully decomposed, and the first step of transitioning to microservice architecture was realized. Additionally, the staff's lack of knowledge and the organization's insufficient infrastructure were identified, facilitating predictions for a transition within the organization.

Key Words: Microservice, Software Architecture, Containerization, Scalability

YÖNTEM

Bu çalışmada, bir yazılım mimarisi olan mikroservis mimarisi incelenmiştir ve monolitik mimari ile oluşturulmuş uygulamanın bir bileşeni bağımsız bir mikroservis olarak tasarlanmıştır. Bu mikroservis, uygulama olarak geliştirilmiş ve monolitik uygulama ile entegre çalışacak şekilde yapılandırılmıştır. Ayrıca SGK'de muhtemel mikroservis uygulamaları için bir ön değerlendirme çalışması yapılmıştır.

İlk aşamada, teorik bir temel oluşturmak için yazılım mimarileri ve tezin ana konusu olan mikroservis mimarisi araştırılmıştır. Bunun için literatür taraması yapılmıştır. Literatür taramasında; kitap, dergi, makale, konferans bildirileri ve bloglardan yararlanılmıştır. Bununla birlikte kurumun mevcut altyapısı hakkında bilgi edinmek için SGK web sitesi kullanılmıştır. Literatür taramasında mikroservislerle ilgili yöntemler ve teknolojiler hakkında bilgi edinilmiştir.

Literatür taramasının ardından monolitik mimariden mikroservis mimarisine geçişin daha kolay ve sorunsuz olması için stratejiler araştırılmış ve geçiş adımları üzerinde durulmuştur. Monolitik uygulamanın analiz edilmesinden bağımsız bir mikroservis olarak geliştirilmesine kadar olan aşamalar tanımlanmıştır.

Belirtilen çalışmalar yapıldıktan sonra SGK'de mikroservis mimarisinin kullanılmasını destekleyen mevcut teknolojiler ve metotlar tespit edilmiştir. Bununla birlikte, geçiş sürecinde karşılaşılabilecek çeşitli zorluklar ve riskler de tanımlanmıştır. Bu riskler arasında sistem karmaşıklığının artması, veri yönetimi sorunları, güvenlik zorlukları ve kaynak kullanımının artması gibi konular yer almaktadır. Ayrıca, SGK'nin mevcut personelinin bu yeni mimariye adaptasyonu için gerekli eğitimler ve DevOps yaklaşımlarının benimsenmesi gerekliliği vurgulanmıştır.

Çalışmanın bir kısmı, SGK'nın teknolojik altyapısının ve insan kaynakları kapasitesinin mevcut durumunu değerlendirmek için bir online anket çalışmasını içermektedir. Anket, tüm BT personeline e-posta yoluyla dağıtılmıştır. Personelin verdiği yanıtların doğruluğunu ve güvenilirliğini sağlamak amacıyla anketin anonim olacağı e-postada ayrıca bildirilmiştir.

Son bölümde, monolitik Emektar4B uygulamasının bir bileşeni ayrıştırılmış ve mikroservis olarak yeniden yazılması detaylı bir şekilde ele alınmıştır. İlk olarak, uygulamanın mevcut durumu ve kullanılan eski teknolojilerin detayları verilmiştir. Mevcut teknolojilerin geçiş sürecine etkisi tartışılmış ve yeni mimariye geçiş için gereken teknolojiler belirlenmiştir. Teknolojilerin seçimi, kurumun mevcut teknoloji altyapısı ve mikroservis mimarisi gereksinimleri göz önünde bulundurularak yapılmıştır. Teknolojilerin kullanımı ve uygulamaların dağıtım süreci de detaylı bir şekilde anlatılmıştır. Sonuç olarak, Emektar4B uygulamasının bir bileşeninin mikroservis mimarisine geçişi için gerekli tüm aşamalar ve teknik detaylar bu bölümde anlatılmıştır.

Tez, önceki bölümlerden toplanan verilerin değerlendirilmesi, karşılaşılan sorunların çözümüne yönelik önerilerin sunulmasıyla tamamlanmaktadır.

İÇİNDEKİLER

BEŞİNCİ BÖLÜM

MİKROSERVİS UYGULAMA ÖRNEĞİ

ŞEKİLLER VE TABLOLAR LİSTESİ

ŞEKİLLER

TABLOLAR

GRAFİKLER LİSTESİ

KISALTMALAR LİSTESİ

4/1-(b)	5510 sayılı Kanunun 4 üncü maddesinin 1 inci fıkrasının (b) bendi
ACID	Atomicity, Consistency, Isolation, Durability (Atomiklik, Tutarlılık, İzolasyon, Kalıcılık)
API	Application Programming Interface (Uygulama Programlama Arayüzü)
CD	Continuous Delivery (Sürekli Teslimat)
CI	Continuous Integration (Sürekli Entegrasyon)
CLI	Command Line Interface (Komut Satırı Arayüzü)
CNCF	Cloud Native Computing Foundation (Bulut Yerel Bilişim Vakfı)
CRUD	Create, Read, Update, Delete (Oluştur, Oku, Güncelle, Sil)
DAO	Data Access Object (Veri Erişim Nesnesi)
DDD	Domain Driven Design (Alan Odaklı Tasarım)
DevOps	Development and Operations (Geliştirme ve Operasyonlar)
ESB	Enterprise Service Bus (Kurumsal Hizmet Veri Yolu)
HTML	Hypertext Markup Language (Hiper Metin İşaretleme Dili)
HTTP	Hypertext Transfer Protocol (Hiper Metin Transfer Protokolü)
IBM	International Business Machines (Uluslararası İş Makineleri Şirketi)
IaaS	Infrastructure as a Service (Hizmet olarak Altyapı)
IP	Internet Protocol (İnternet Protokolü)
IT	Information Technologies (Bilgi Teknolojileri)
JSP	JavaServer Pages (JavaServer Sayfaları)
MVC	Model View Controller (Model Görünüm Denetleyici)
ORM	Object Relational Mapping (Nesne ile İlişkisel Eşleme)
PaaS	Platform as a Service (Hizmet olarak Platform)
REST	Representational State Transfer (Temsili Durum Transferi)
SaaS	Software as a Service (Hizmet olarak Yazılım)
SGK	Sosyal Güvenlik Kurumu
SOA	Service-Oriented Architecture (Servis-Odaklı Mimari)
SQL	Structured Query Language (Yapılandırılmış Sorgulama Dili)
SSL	Secure Sockets Layer (Güvenli Soket Katmanı)
URI	Uniform Resource Identifier (Tekdüzen Kaynak Tanımlayıcı)
URL	Uniform Resource Locator (Tekdüzen Kaynak Bulucu)
XML	Extensible Markup Language (Genişletilebilir İşaretleme Dili)

GİRİŞ

Günümüzün hızla gelişen ve değişen teknoloji ortamında yazılım geliştirme süreç ve yöntemleri ihtiyaçlara cevap verebilmek için sürekli bir dönüşüm içindedir. Hız, performans ve depolama kapasiteleri artan bilgisayar sistemleri daha karmaşık ve güçlü yazılımların geliştirilmesine olanak tanımıştır. Bunun sonucunda üretilen büyük ölçekli yazılımları geliştirmek ve yönetmek için yeni mimari yaklaşımlar ortaya çıkmıştır.

Klasik yazılım geliştirme mimarisinde tek bir kod tabanı üzerinde geliştirme yapılır ve geliştirilen yazılım bir bütün halinde kullanıma alınır. Özellikle büyük ölçekli yazılımların geliştirilmesi ve yönetilmesi konusunda bu yaklaşım bazı sınırlamaları ve zorlukları beraberinde getirir. Bu mimarinin getirdiği zorlukları aşmak ve daha esnek, ölçeklenebilir ve dağıtılabilir yazılım çözümleri sunabilmek için farklı bir mimari model olan mikroservis mimarisi ortaya çıkmıştır.

Mikroservis mimarisinin kullanımı Netflix, Amazon, Spotify gibi dünyadaki önemli kurumsal şirketlerin bu mimariyi kullanmada öncülük etmesiyle yazılım dünyasında yaygınlaşmıştır (Cebeci & Korçak, 2020). Bu mimari büyük monolitik uygulamaların yerine küçük, bağımsız ve birbiri ile entegre çalışabilen servislerden oluşan bir uygulama yapısı önermektedir. Her bir mikroservis, belirli bir işlevi gerçekleştirir ve bu servisler bir araya gelerek büyük uygulamaları oluşturur.

SGK bünyesinde hizmet veren yazılımlar, milyonlarca insanın hayatını doğrudan etkileyen kritik öneme sahip yazılımlardır. Bu sistemlerin doğru bir şekilde tasarlanması ve geliştirilmesi hem kurumun işleyişi hem de hizmet verilen vatandaşlar için büyük önem taşımaktadır. Büyük uygulama yazılımlarının karmaşıklığını azaltmak, uygulamaları sıkça güncelleyebilmek ve yazılım ekiplerinin rahat ve uyumlu bir şekilde çalışmasını sağlamak için sunulan bir çözüm olan mikroservis mimarisi, kurum yapısına uygunluğu ve kullanılabilirliği açısından değerlendirilecektir. Avantajları ve dezavantajları göz önüne alınarak ne kadar optimize bir çözüm olduğu hem teknik açıdan hem kültürel açıdan ele alınacaktır.

BİRİNCİ BÖLÜM
YAZILIM MİMARİSİ

1. TANIM

Yazılım mimarisi, bir yazılım sisteminin oluşturulması için gerekli olan yapıların ve bu yapıları oluşturmak için izlenen disiplinin bütünüdür. Yazılım mimarisi, yazılım bileşenlerini, bu bileşenler arasındaki ilişkileri ve hem bileşenlerin hem de ilişkilerinin özelliklerini içerir (Bass, et al., 2012, p. 3).

Yazılımlar belirli bir işi yerine getirmek için yapılır. Yazılım planının soyut bir hedef halinden bir işleme çözüm getiren somut bir hale gelmesi kompleks bir süreçtir. Yazılım mimarisi, süreçleri yönetebilmek ve iş hedeflerini gerçekleştirebilmek için teorik bir çerçeve sağlar (Bass, et al., 2012, p. 2). Bu çerçeve, iş gereksinimlerini, kullanıcı ihtiyaçlarını ve teknik kısıtlamaları dikkate alarak somut ürünün meydana gelmesinde kritik bir rol oynar.

2. YAZILIM MİMARİSİNİN YARARLARI

Yazılım mimarisinin sağladıkları yararlar aşağıdaki gibidir:

- ✓ Yazılım sisteminin temel yapısını ve bileşenlerinin birbiriyle nasıl etkileşime girdiğini tanımlar. Tüm paydaşlar için daha anlaşılabilir hale getirir (Bass, et al., 2012, p. 2).
- ✓ Süreç yönetimi ve maliyet tahmini yapılmasını sağlar (Perry & Wolf, 1992).
- ✓ Paydaşlar arasındaki iletişimi güçlendirir (Bass, et al., 2012, p. 21).
- ✓ Yapılan yazılım mimarisinin parçaları farklı sistemlerde yeniden kullanılabilir. Bu geliştirme süresini ve maliyetini kısaltmaya yardımcı olur (Bass, et al., 2012, p. 21).
- ✓ Yazılım gelecekteki muhtemel değişikliklere uyum sağlayabilecek bir yapıda geliştirilmesini sağlar.
- ✓ Yazılımın esnekliğini ve ölçeklenebilirliğini artırır. Böylece yazılımın büyütülebilmesine ve değiştirilebilmesine olanak sağlar.

3. YAZILIM MİMARİSİ ÇEŞİTLERİ

Yazılım mimarisinin evrimi, değişen teknolojik gereksinimler ve iş zorluklarına cevap olarak gelişmiştir. Yazılım mimarisi literatüründe adı geçen üç mimari stili incelenecektir.

3.1. Monolitik Mimari

Monolitik mimari, yazılım geliştirme dünyasında uzun süredir kullanılan geleneksel bir mimari yaklaşımdır. Bu yaklaşım, yazılımın tüm fonksiyonlarının tek bir birim olarak geliştirilmesini ve dağıtılmasını ifade eder (Chen, et al., 2017, pp. 466-475). Bu yaklaşımda kullanıcı arayüzü, iş mantığı, veri erişim katmanı ve uygulama entegrasyonu işlevlerinin tümü tek parça halindedir. Tek bir kod tabanı üzerine kurulu bu sistemler, başlangıçta geliştirilmesi ve anlaşılması kolay sistemlerdir. Çünkü tüm bileşenler tek bir programlama dilinde ve çalışma zamanı ortamında geliştirilir.

Zamanla yazılımın ihtiyaçları karşılaması için yapılan değişiklikler ve eklemelerle yazılımın boyutu ve kod karmaşıklığı artar. Bu durum yazılımın ölçeklendirme ve sürekli entegrasyon gibi ihtiyaçları karşılayamamasına neden olur. Kodda yapılan herhangi bir değişiklik bütün yapının yeniden konuşlandırılmasını(deploy) gerektirir. Ayrıca, yazılımın herhangi bir yerindeki hata tüm yazılımın çalışmamasına neden olabilir. Bu durum hata izolasyonunun ve sistem esnekliğinin olmaması demektir.

Bu mimari yaklaşımın özellikle daha küçük yazılımlar için veya bir yazılımın boyutu ve kapsamı nispeten sabit ve yönetilebilir olduğunda çeşitli avantajları da vardır. Tüm bileşenler aynı uygulamada olduğundan uygulamanın farklı bölümleri arasındaki iletişim hızlıdır. Hata ayıklama ve test etme işlemleri basittir. Harici hizmetlere herhangi bir bağımlılık olmaksızın yönetilecek tek bir uygulama olduğundan konuşlandırma(deploy) kolaydır.

3.2. Servis Odaklı Mimari

Servis-odaklı mimari (SOA), sistem bileşenlerinin bağımsız hizmetler olduğu dağıtılmış sistemler geliştirmenin bir yoludur (Sommerville, 2010). Bileşenler birbirleriyle ağ üzerinden bir iletişim protokolü ile haberleşir. SOA'da temel amaç işlevleri farklı servislere ayırmaktır.

SOA kullanımının yazılım sistemlerine esnek, verimli ve güvenli bir şekilde çalışabilmesi için getirdiği bazı avantajlar vardır. Bunlar:

- ✓ SOA'da her servis spesifik bir işlevi yerine getirmek üzere oluşturulur. Bu servisler gerektiğinde yeniden kullanılabilir. Bu da iş ve zamandan tasarruf edilmesini sağlar.
- ✓ Servisler işlevlerini yerine getirirken iç uygulamanın mantığını yani nasıl yaptıklarını kullanıcıdan gizlerler. Kullanıcıya basit bir arayüz sunarak karmaşıklığı ortadan kaldırır.
- ✓ Servisler birbirinden bağımsız olarak geliştirilebilir ve konuşlandırılabilir.
- ✓ Değişen iş gereksinimlerini karşılamak için gelişmiş esneklik sağlar. Yeni servisler eklenebilir, var olan servisler güncellenebilir ve iş süreçleri yeniden yapılandırılabilir.
- ✓ Farklı sistemlerin birlikte çalışabilmesi ve veri paylaşması kolaydır.
- ✓ Modüler bir yapıya sahiptir ve sistem bileşenleri izole bir şekilde çalıştığından güncellenmesi durumunda birbirinden etkilenmez.
- ✓ Ölçeklenebilirliği sayesinde artan kullanıcı sayısı ve/veya iş yükü durumunda sistem performansını korumak için sistem büyütülebilir.

Bunun yanı sıra SOA bazı dezavantajları da beraberinde getirir. Bunlar:

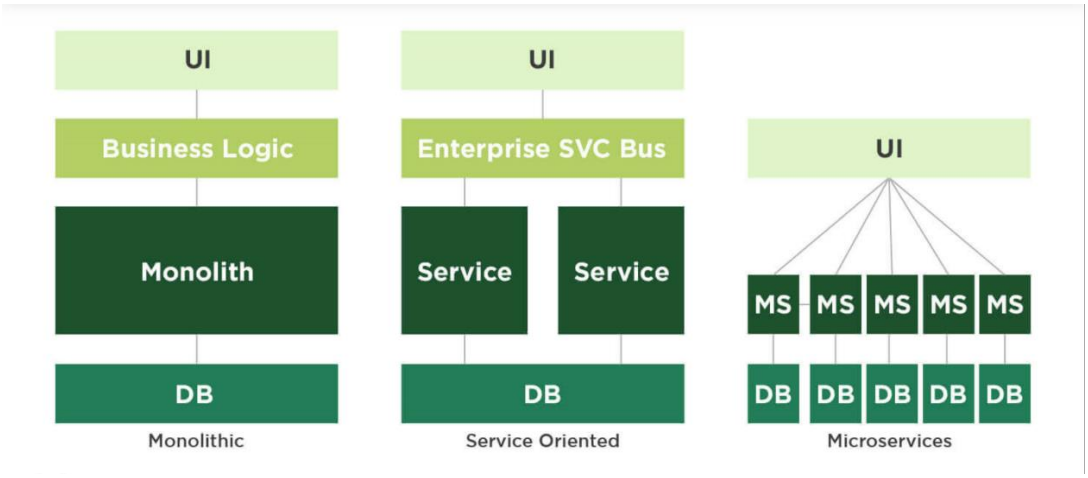
- ✓ Ağ performansı ve servisler arası iletişim gecikmeleri sistemin çalışma performansını yavaşlatabilir.
- ✓ Sistem birçok servis içermesi durumunda bu servisler arasındaki iletişimi yönetmek ve izlemek zor olabilir. Bu karmaşıklık hataları tespit etmeyi ve çözmeyi zorlaştırır (Jayasooriya, 2024).
- ✓ Birçok farklı servis bağlantı noktası içerdiğinden güvenlik riskleri doğar. Servislerin arasındaki iletişimin güvenliğini sağlamak ek güvenlik önlemleri gerektirir.

3.3. Mikroservis Mimarisi

Mikroservis mimarisi, yazılım geliştirme dünyasında modern ve esnek bir mimari olarak öne çıkmıştır. Bu mimaride, büyük ve karmaşık yazılım uygulamaları, birbirinden bağımsız olarak çalışabilen küçük, otonom hizmetlere bölünür. Her

mikroservis, belirli bir işlevi yerine getirir ve diğer mikroservislerle iletişim halindedir. Bütün mikroservisler genel sistemin bir parçasıdır. Bu yaklaşım, büyük monolitik yapılara göre daha esnek ve ölçeklenebilirdir.

Şekil 1: Yazılım Mimarilerinin Evrimi



Kaynak: (Ilyukha, 2024)

Mikroservislerin en büyük avantajlarından biri, bağımsız olarak geliştirilip dağıtılabilmeleridir. Her mikroservis, farklı teknoloji ile geliştirilebilir ve birbirinden bağımsız olarak güncellenip ölçeklenebilir. Bu, özellikle büyük ekiplerin paralel çalışmasını ve yazılım geliştirme süreçlerinin hızlanmasını sağlar. Ayrıca, bir mikroservisin başarısız olması durumunda, bu sadece o hizmeti etkiler ve tüm sistemin çökmesini engeller, bu da sistem güvenilirliğini artırır.

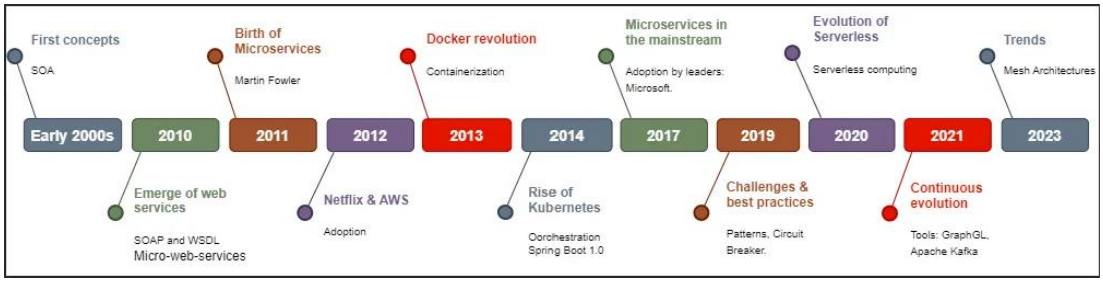
Mikroservis mimarisi tezin ana konusu olduğundan bir sonraki bölümde daha detaylı bir şekilde anlatılmıştır.

İKİNCİ BÖLÜM
MİKROSERVİS MİMARİSİ

1. MİKROSERVİS MİMARİSİ TARİHÇESİ

Mikroservis terimi yazılım dünyasına çeşitli aşamalardan geçerek gelmiştir. Dr. Peter Rodgers, 2005 yılında "Micro-Web-Services" terimini ilk kullanan kişi olarak kabul edilmektedir. "Mikroservis" terimi ise 2014 yılında Martin Fowler ve ThoughtWorks'teki meslektaşlarının bu kavramı sağlam ve ölçeklenebilir sistemler oluşturmanın bir yolu olarak tartışmasıyla önem kazanmıştır (Sanchez, 2024).

Şekil 2: Mikroservis Mimarisinin Gelişimini Etkileyen Dönüm Noktaları



Kaynak: *(Sanchez, 2024)*

Mikroservis mimarisi fikrinin temelleri 2000'li yılların başlarına kadar gider. Monolitik mimarilerin uygulamalar büyüdükçe karmaşıklığının artması uygulamaların birbirinden ayrı olarak geliştirilip bir protokol üzerinden haberleşmesi fikrinin yani SOA fikrinin doğmasına yol açmıştır.

2010'larda Amazon ve Netflix gibi dünya çapındaki şirketlerin çevikliği ve ölçeklenebilirliği artırmak için büyük monolitik uygulamalarını daha küçük ve bağımsız olarak dağıtılabilen servislere ayırması mikroservis mimarisinin popülaritesini artırdı.

2013'te Docker ve 2014'te Kubernetes teknolojilerinin çıkması mikroservis mimarisinin kullanımına kolaylıklar getirdi. Docker her bir mikroservisin hafif konteynerler olarak paketlenmesine olanak tanırken Kubernetes daha kolay bir biçimde mikroservislerin dağıtımlarının yönetilmesini sağladı. Bu araçların

geliştirilmesi ve kullanımının yaygınlaşması mikroservis mimarisinin yazılım dünyasında yerini daha da sağlamlaştırmasına neden oldu.

Sonraki yıllardan günümüze kadar gelen süreçte mikroservis mimarisinin kullanımı artmaya ve büyük ölçekli web uygulamaları için standart haline gelmeye devam etmektedir. DevOps uygulamalarının ve sürekli teslimatın gelişimi mikroservis mimarisinin yazılım dünyasındaki yerini desteklemektedir.

2. MİKROSERVİS MİMARİSİ TANIMI VE ÖZELLİKLERİ

Mikroservis mimarisi, her biri kendi sürecinde çalışan ve hafif mekanizmalarla haberleşme sağlayan küçük servislerden oluşan tek bir uygulamayı geliştirmeye yönelik bir yaklaşımdır (Fowler & Lewis, 2014). Başka bir şekilde tanımlanırsa, tek bir uygulamayı küçük, özerk servislerden oluşan bir koleksiyona ayıran merkezi olmayan bir yazılım mimarisidir.

Mimariyi oluşturan her bir servis tek sorumluluk prensibine uyarak kendi için tanımlanmış olan göreve odaklanır. Servisler kendi işine odaklanan parçalara bölündüğü için küçüktür. Birbirleriyle haberleşirken HTTP gibi teknolojiden bağımsız protokoller kullanırlar.

Mikroservis mimarisinin kendine özgü nitelikleri aşağıdaki şekilde sıralanabilir:

- ✓ Servisler sadece tek bir iş yapmak üzerine tasarlandığı için küçüktür.
- ✓ Her bir servis büyük ölçüde özerktir. Başka bir ifade ile servisler birbirlerine gevşek bağlıdır ve birbirlerinin süreçlerine müdahale edemezler. İzole bir şekilde çalışırlar.
- ✓ Mikroservisler ihtiyaca göre birbirlerinden bağımsız bir şekilde ölçeklendirilebilir.
- ✓ Teknoloji bağımsızdır. Her bir servis farklı programlama dilini, farklı bir çerçeveyi(framework) ve veritabanını kullanabilir.
- ✓ Mikroservis mimarisi karmaşıktır ve yönetilmesi zor bir yapıya sahiptir.

2.1. Mikroservis Mimarisinin Avantajları ve Dezavantajları

Mikroservis mimarisi monolitik mimaride ortaya çıkan zorlukların üstesinden gelmek için oluşturulan bir çözümdür. Artı yönleri aşağıda sıralanmıştır:

- ✓ Mikroservis mimarisi, her servis için yerine getireceği göreve uygun programlama dilini ve araçlarını seçme esnekliği sağlar ve yenilikçiliği teşvik eder.
- ✓ Mikroservislerden herhangi biri arızalanırsa diğer servisler çalışmaya devam ederek sistemin genel hata toleransını artırır.
- ✓ Uygulamanın tamamını ölçeklendirmek yerine ihtiyaca göre sadece belirli bir servisi ölçeklendirme olanağı sunar.
- ✓ Mikroservis mimarisi ile tek bir serviste değişiklik yapılabilir ve sitemin geri kalanından bağımsız olarak dağıtılabilir. Hızlı dağıtıma olanak sağlar (Newman, 2015).
- ✓ Herhangi bir kod tabanında çalışan kişi sayısını en aza indirmek ekip büyüklüğü ve üretkenlik gibi önemli parametreler için uygun seviyeyi yakalamayı kolaylaştırır (Newman, 2015).

Bunların yanında mikroservis mimarisi kendine özgü eksi yönleri de içinde barındırır. Bunların aşağıdaki gibidir:

- ✓ Birden fazla servisi yönetmek ve düzenlemek karmaşıktır. Ek araç kullanımı gerektirir.
- ✓ Mikroservisler ağ üzerinden iletişim kurar bu sistemin çalışmasında yavaşlamaya neden olabilir (Fowler & Lewis, 2014).
- ✓ Veri tutarlılığını sağlamak zordur. Potansiyel veri bütünlüğü sorunlarına yol açar (Kızılpınar, 2021).
- ✓ Mikroservisler doğası gereği dağıtık yapıda oldukları için hata ayıklama ve test işlemleri zordur.
- ✓ Mikroservisler daha fazla potansiyel giriş noktasına sahip olduğu için saldırıya daha açıktır.

2.2. Mikroservis Mimarisinin Monolitik Mimari ile Karşılaştırılması

Mikroservis mimarisi geleneksel yazılım geliştirme yaklaşımı olan monolitik mimarinin zayıflıklarını gidermek için oluşturulmuştur. İki yaklaşımın farkları karşılaştırması şu şekildedir:

- ✓ Mikroservis mimarisinde ölçeklendirme daha kolaydır. Uygulamanın sadece belli bir bölümü ölçeklendirilebilir. Monolitik mimaride ise uygulama bir bütün halinde ölçeklendirilmek zorundadır. Bu da fazladan maliyet getirir (Ayrancıoğlu, 2019).
- ✓ Mikroservis mimarisinde yapılan bir değişiklik sadece belli bir servisi etkilerken monolitik mimaride tüm uygulama yeniden konuşlandırma yapılması gerekliliği doğar. Güncelleme monolitik mimaride daha zordur.
- ✓ Mikroservis mimarisinde yeni bir teknolojiye geçmek ve kullanmak kolayken monolitik mimaride tüm uygulamanın geçirilmesi gerektiğinden daha zordur.
- ✓ Mikroservis mimarisinde hata ayıklama ve test etme işlemleri ek araçlar gerektirir ve zahmetlidir. Fakat monolitik mimaride daha basittir.
- ✓ Mikroservis mimarisinde servisler arası haberleşme ağ üzerinden yapılır ve bu ek yük getirir. Monolitik mimaride böyle bir sorun yoktur.

2.3. Mikroservis Mimarisinin SOA ile Karşılaştırılması

SOA ve mikroservis mimarileri, yazılım geliştirme sürecine çeviklik ve esneklik sağlamayı amaçlayan mimari desenlerdir ancak yaklaşımları bakımından belli farklılıkları vardır. Bunlar aşağıdaki gibidir:

- ✓ Mikroservisler, tipik olarak daha büyük SOA servislerine göre daha spesifik olacak şekilde tasarlanmıştır. Yani tek bir işleve veya birbiriyle yakından ilişkili küçük bir işlevsellik kümesine odaklanır.
- ✓ Mikroservisler sıklıkla kendi veritabanına sahiptir, SOA servisleri veritabanlarını paylaşır.
- ✓ Mikroservisler haberleşme için REST veya mesajlaşma kuyrukları gibi daha hafif protokoller kullanırken SOA, ESB aracılığıyla haberleşir.
- ✓ Mikroservisler bağımsız bir şekilde konuşlandırılabilirken SOA servisleri genellikle eşzamanlı bir şekilde konuşlandırılır.

- ✓ Mikroservisler dağıtık yapısı nedeniyle doğası gereği ölçeklenebilirler.

Projenin boyutu ve karmaşıklığı, beklenen ölçeklenebilirlik kapasitesi, organizasyon becerileri ve mevcut altyapısı göz önünde bulundurarak mimari ele alınmalıdır.

3. MİKROSERVİS MİMARİSİYLE İLİŞKİLİ KAVRAMLAR

Mikroservis mimarisi diğer yazılım metotları ve teknolojileri ile birlikte çözüm getirir. Alt başlıklarda mikroservis mimarisinin tasarımında ve çalıştırılmasında ilişkili olduğu kavramlar ve teknolojilerden bahsedilecektir.

3.1. Alan Odaklı Tasarım (DDD)

DDD, hizmet etmeyi amaçladığı iş alanına sıkı sıkıya bağlı yazılım sistemleri oluşturmaya odaklanan bir yazılım geliştirme metodolojisi ve mimari yaklaşımdır (Lotfy, 2023). DDD'de ilk amaç yazılım paydaşlarının ortak bir iletişim dili oluşturmasıdır. İkinci amacı ise ölçeklenebilir ve anlaşılabilir bir yazılım mimarisi oluşturmaktır.

İlk kez Eric Evans'ın kitabında sözü geçen terim yazılım sisteminin temel alanının anlaşılmasını ve modellenmesini vurgular. Oluşturulacak yazılıma ilişkin ortak bir anlayış oluşturmak için yazılıma katkıda bulunacak paydaşların arasındaki iş birliğini teşvik eder. DDD etki alanına odaklanarak geliştiricilerin gerçek dünyadaki iş gereksinimleriyle yakından uyumlu yazılım çözümleri oluşturmasına olanak tanır.

DDD, herkesin etki alanı kavramlarını anlamasını sağlamak için yazılım geliştiricileri ve iş dünyası arasında ortak bir kelime dağarcığı olan her yerde bulunan dil(Ubiquitous Language) oluşturulmasını savunur. Etki alanını kendi içinde ilişkili parçalara böler.

DDD, sınırları belirlemek ve sistemi bölmek için uygun bir çözüm sağlar (Pachghare, 2016). DDD, etki alanlarını net bir şekilde arayüzlerle birbirinden ayırdığı için yazılımların dağıtık bir şekilde geliştirilmesine olanak tanır. Mikroservis mimarisi dağıtık bir mimari olduğu için DDD ile tasarlanan yazılımlar her bir etki alanı bir mikroservis olacak şekilde ayrı şekilde konuşlandırılmaya elverişlidir.

3.2. DevOps

DevOps, geliştirme ve operasyon dünyalarını otomatik geliştirme, dağıtım ve altyapı izleme ile bütünleştiren bir yazılım pratiğidir (Ebert, et al., 2016). Geleneksel yazılım geliştirme süreçlerinde, geliştirici ekipler ve operasyon ekipleri genellikle ayrı çalışır ve bu durum, yazılımın geliştirilmesi ile üretim ortamına aktarılması arasında gecikmelere ve uyumsuzluklara neden olabilir. DevOps, bu iki dünyayı bir araya getirerek, geliştirme ve operasyon ekiplerinin daha sıkı bir iş birliği içinde çalışmasını sağlar. Bu sayede, yazılım projelerinin daha hızlı, daha güvenilir ve daha sürdürülebilir bir şekilde tamamlanmasını mümkün kılar. DevOps'un en önemli özelliklerinden biri, sürekli entegrasyon (CI) ve sürekli teslimat (CD) süreçlerini kullanarak, yazılım değişikliklerinin anında test edilip üretim ortamına aktarılmasını sağlamasıdır. DevOps sayesinde, yeni özellikler ve iyileştirmeler hızla devreye alınır.

Mikroservis mimarisinin etkin bir şekilde uygulanabilmesi için DevOps pratiğinin kullanılması çok önemlidir. Mikroservis mimarisi ile DevOps'un entegrasyonunun gerekliliği üzerine bazı temel nedenler şu şekildedir:

CI/CD: Mikroservis mimarisi, hizmetlerin bağımsız olarak geliştirilmesine, test edilmesine ve dağıtılmasına olanak tanır. DevOps, CI ve CD araçlarıyla bu süreçleri otomatize ederek, kod değişikliklerinin hızlı ve güvenli bir şekilde üretime aktarılmasını sağlar. CI/CD, mikroservislerin sorunsuz güncellemelerini destekler. Bu sayede ekipler arası bağımlılıkları azaltır ve dağıtım süreçlerini hızlandırır.

Otomasyon ve Orkestrasyon: Mikroservisler genellikle konteynerler içinde paketlenir ve çalıştırılır. DevOps, Docker ve Kubernetes gibi araçlar ile entegre çalışarak, mikroservislerin ölçeklenmesi, yönetimi ve izlenmesi işlemlerini otomatize eder. Bu araçlar, mikroservislerin dağıtımını, ölçeklenmesini ve hata yönetimini basitleştirir ve otomatikleştirir.

Hız ve Esneklik: DevOps yaklaşımı, hızlı geliştirme ve esneklik prensipleri üzerine kuruludur. Mikroservis mimarisi de benzer şekilde, hızlı ve dinamik değişikliklere uyum sağlama yeteneği sunar. DevOps'un sürekli test ve geribildirim döngüleri, mikroservislerin hızlı bir şekilde geliştirilmesini ve potansiyel sorunların erken aşamada tespit edilip çözülmesini sağlar.

İzleme ve Güvenlik: Mikroservis mimarisi, çok sayıda bağımsız hizmetin bir arada çalıştığı karmaşık sistemler oluşturur. DevOps, loglama, izleme ve güvenlik uygulamaları ile bu hizmetlerin performansını sürekli izler ve güvenlik tehditlerine karşı korur. Bu, sistem genelindeki hataların, performans sorunlarının ve güvenlik açıklarının yönetilmesini sağlar.

Kültürel Uyum: DevOps, iş birliği ve iletişim üzerine kurulu bir kültürü teşvik eder. Mikroservis mimarisi, farklı ekipler tarafından bağımsız olarak geliştirilen hizmetleri içerdiğinden, ekipler arası koordinasyon ve iş birliği için güçlü bir kültürel yapı gereklidir. DevOps, bu kültürü destekleyerek, ekiplerin daha uyumlu ve etkili bir şekilde çalışmasını sağlar.

Mikroservis mimarisi, DevOps pratiği olmadan teorik olarak uygulanabilir olsa da DevOps'un sunduğu süreçler, araçlar ve kültürel yaklaşımlar, bu mimarinin etkin, verimli ve başarılı bir şekilde uygulanabilmesini sağlar. DevOps, mikroservislerin potansiyelini tam olarak ortaya çıkarmada ve modern yazılım geliştirme hedeflerine ulaşmada önemli bir yaklaşımdır.

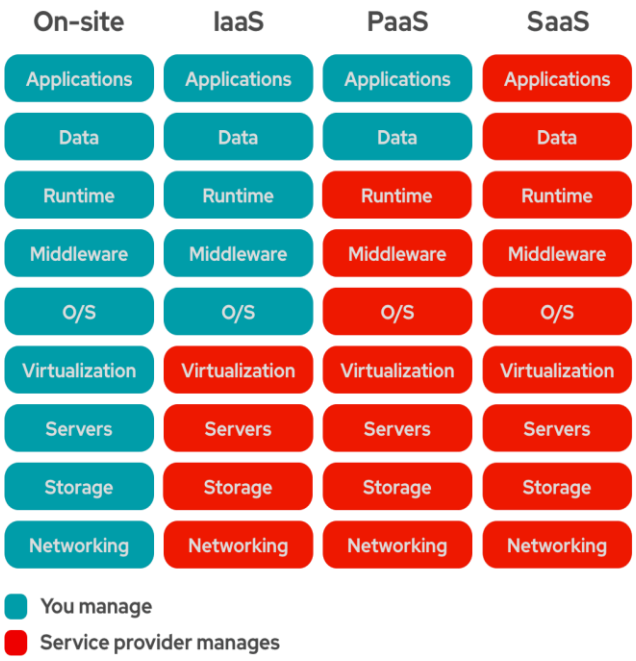
3.3. Bulut Bilişim (Cloud Computing)

Bulut bilişim, minimum yönetim çabası veya hizmet sağlayıcı etkileşimi ile hızlı bir şekilde sağlanabilen ve serbest bırakılabilen yapılandırılabilir bilgi işlem kaynaklarının paylaşılan bir havuzuna her yerde, uygun, isteğe bağlı ağ erişimi sağlamaya yönelik bir modeldir (Mell & Grance, 2011). Bulut bilişim ile birlikte bilgi teknolojileri kaynaklarına erişim, kullanım ve yönetim biçimleri değişmiştir. Çeşitli hizmetlerin internet üzerinden sunulmasını sağlamaktadır. Böylece bu hizmetten yararlananlar fiziksel sunuculara ve diğer altyapılara sahip olma ve bunların bakımını yapma masraflarından ve karmaşıklığından kurtulurlar.

Bulut bilişim her biri farklı ihtiyaçlara ve senaryolara hitap edecek şekilde üç hizmet modeli ortaya çıkmıştır: Bunlar; Hizmet Olarak Altyapı (IaaS), Hizmet Olarak Platform (PaaS) ve Hizmet Olarak Yazılım (SaaS). Her model farklı bir soyutlama

seviyesini temsil eder ve çeşitli bilgi işlem senaryolarında farklı kullanıcı ihtiyaçlarını karşılar.

Şekil 3: Bulut Bilişim Hizmet Modelleri



Kaynak: *(Red Hat, 2022)*

IaaS, bulut hizmetinin en temel seviyesidir. Kullanıcılara sunucu, ağ, sanallaştırma ve depolama gibi temel bilgi işlem altyapısına erişim sunarak kurumların pahalı donanımlara yatırım yapmak yerine kaynakları talep üzerine ve ihtiyaç duyulduğunda satın almasına olanak tanır. Kurumların talebe göre kaynakları hızlı bir şekilde ölçeklendirmelerine veya azaltmalarına olanak tanır. IaaS aynı zamanda fiziksel sunucuları ve diğer veri merkezi bileşenlerini yönetmenin karmaşıklığından ve maliyetinden kaçınmak isteyen kurumlar için de çok önemli bir teknolojidir.

PaaS kurumların süreçle ilişkili altyapıyı oluşturma ve sürdürme karmaşıklığı olmadan uygulamaları geliştirmelerine, çalıştırmalarına ve yönetmelerine olanak tanıyan bir platform sağlayan bulut hizmeti modelidir. PaaS, işletim sistemleri, geliştirme araçları, veritabanı yönetim sistemleri ve daha fazlasını içerir ve web uygulaması yaşam döngüsünün tamamını (oluşturma, test etme, dağıtma, yönetme ve güncelleme) destekler. PaaS, temel altyapıyı yönetmek zorunda kalmadan uygulama

geliştirmenin yaratıcı yönüne odaklanmak isteyen geliştiriciler için özellikle faydalıdır.

SaaS, bulut bilişimin en yaygın olarak bilinen şeklidir. SaaS, yazılım uygulamalarını abonelik temelinde İnternet üzerinden sunar. SaaS sağlayıcıları altyapıyı, platformları ve hatta verileri yönetir; bu da kullanıcıların uygulamaları tek tek bilgisayarlara yüklemelerine veya çalıştırmalarına gerek olmadığı anlamına gelir. Bu sadece yazılım edinme masraflarını azaltmakla kalmaz, aynı zamanda bakım ve desteği de basitleştirir. Yaygın örnekleri arasında e-posta, müşteri ilişkileri yönetimi sistemleri ve iş birliğine dayalı araçlar yer almaktadır. Bu model, hızlı kurulum ve dağıtım, maliyet etkinliği ve ölçeklenebilirlik açısından avantajlıdır ve İnternet bağlantısı ve tarayıcısı olan herhangi bir cihazdan erişilebilen çözümler sunar.

Bu hizmet modelleri, altyapı tedarikinden uygulama geliştirme ve yazılım sunumuna kadar farklı ihtiyaçlara hitap eden bulut bilişim hizmetleri yelpazesini temsil etmektedir. Bu modeller, bulut bilişimin azaltılmış sermaye giderleri, daha düşük operasyonel maliyetler, gelişmiş ölçeklenebilirlik, gelişmiş erişilebilirlik ve daha iyi yönetilebilirlik gibi faydalarını kapsamaktadır. Bu modelleri anlamak, operasyonlarını ve stratejik yeteneklerini geliştirmek için bulut teknolojilerinden yararlanmak isteyen kuruluşlar için çok önemlidir.

Bulut ortamları, ölçeklenebilirlikleri, esneklikleri ve dağıtılmış yapıları ile mikroservis mimarisi için ideal bir altyapı sağlar. Yükü dengelemek ve her bir mikroservisin optimum performansını sağlamak için çok önemli olan dinamik kaynak tahsisini ve yönetimini desteklerler. Mikroservis mimarisi ve bulut bilişim teknolojisi birbirini tamamlayan bir yapıya sahiptir.

3.4. Docker

Yazılım konteynerizasyonunda devrim yaratan açık kaynaklı bir platform olan Docker, konteynerleri kullanarak uygulamaların oluşturulmasını, dağıtılmasını ve çalıştırılmasını basitleştirir. Konteynerler, bir geliştiricinin bir uygulamayı kütüphaneler ve diğer bağımlılıklar gibi ihtiyaç duyduğu tüm parçalarla birlikte paketlemesine ve hepsini tek bir paket olarak göndermesine olanak tanır. Docker bunu yaparak öngörülebilirliği ve verimliliği artırır. Uygulamaların ayrılmış ve kontrollü bir ortamda çalışmasını sağlar.

Docker hem geliştirme hem de dağıtım iş akışlarını verimli, ölçeklenebilir ve güvenli bir şekilde yapmak için hafif konteynerleştirme teknolojisini kullanır. Teknoloji, Docker konteynerleri içinde çalışan uygulamaların birbirinden ayrı ve izole olmasını sağlamak için birden fazla güvenlik katmanını destekler, bu da kötü niyetli saldırıların veya sistem arızalarının bitişik konteynerleri etkileme riskini en aza indirir.

Sonuç olarak Docker, farklı ortamlarda çok çeşitli uygulamaları yönetmek için ölçeklenebilir, güvenli ve verimli bir platform sağlayarak yazılım geliştirme ve dağıtım alanında önemli bir ilerlemeyi temsil etmektedir. Docker, CI/CD iş akışları için ideal bir çözümdür; çünkü geliştirme, test ve üretim ortamları arasında tutarlılık sağlar. Ayrıca uygulamaları dağıtık olarak yönetmeye elverişli olduğundan mikroservis mimarisiyle uyumludur. Bununla birlikte, Docker'ın getirdiği yenilikler ve esneklikler, büyük ölçekli sistemlerde kaynak kullanımını optimize etmek ve yönetmek için de önemli fırsatlar sunmaktadır.

3.4.1. Docker'ın Temel Bileşenleri

Docker'ın mimarisi, dağıtılmış uygulamaları çalıştırmak için hafif, taşınabilir ve verimli bir araç sağlamak üzere birlikte çalışan birkaç temel bileşenden oluşur. Docker'ın temel bileşenleri aşağıda incelenecektir ve bu bileşenlerin Docker ekosistemi içindeki işlevleri ve rolleri açıklanacaktır.

Docker Engine: Docker'ın işlevselliğinin merkezinde, Docker konteynerlerinin oluşturulmasını ve güvenliğini sağlayan hafif bir çalışma zamanı ve paketleme aracı olan Docker Engine yer almaktadır. Bu teknoloji istemci-sunucu tabanlıdır. Üç ana bileşeni vardır: Konteynerleri yöneten dockerd, dockerd ile iletişim sağlayan REST API'lar ve komut satırı.

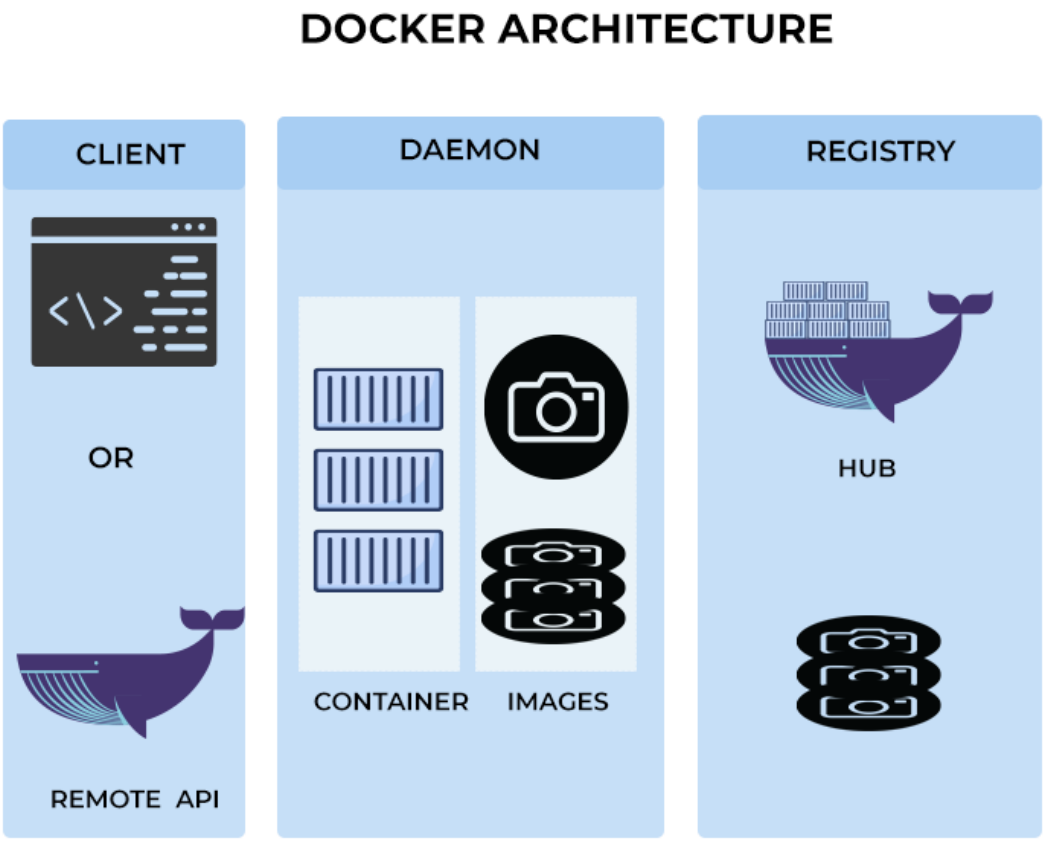
Docker Image: Konteyner oluşturmak için kullanılan salt okunur bir şablondur (docker.docs, 2024). Kaynak kodlarını, kütüphaneleri ve bağımlılıkları içerir. Docker Hub veya özel kayıt defterleri(registry) üzerinden indirilebilir. Oluşturulan Docker Image'lar Docker Hub sitesine yüklenebilir.

Docker Container: Konteyner, bir Docker Image'ının çalıştırılabilir bir örneğidir (docker.docs, 2024). Docker API veya CLI ile konteynerleri başlatmak, durdurmak ve yönetmek mümkündür. Bir Docker Image çalıştırıldıktan sonra belleğe alınır.

Docker Registry: Docker Image'larının depolandığı yerdir. Docker Hub herkesin kullanabileceği bir Docker Registry'dir.

Docker Client: Kullanıcı ile etkileşimi sağlayan bileşendir. Bir CLI ile bunu gerçekleştirir.

Şekil 4: Docker Bileşenleri

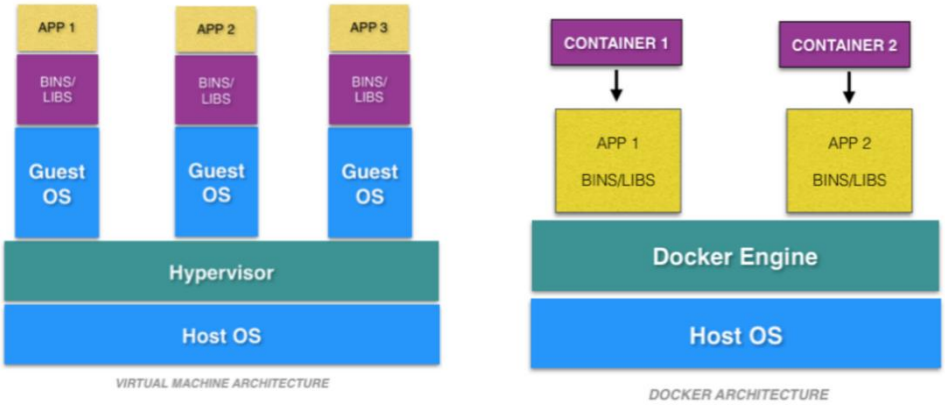


Kaynak: (Basumallick, 2022)

3.4.2. Sanal Makine ve Docker Karşılaştırması

Konteynerizasyon teknolojisi Docker ile sanal makine teknolojileri hem taşınabilirlik hem de izolasyon sağlayan sanallaştırma teknolojileridir. Her ikisi de donanım kaynaklarının daha verimli bir biçimde kullanılmasını sağlarlar. Bununla birlikte birbirlerine karşı avantajlara ve dezavantajlara sahiptir.

Şekil 5: Docker ve Sanal Makine Mimarileri



Kaynak: *(Cloud Academy Team, 2023)*

3.3.2.1. Mimari

Sanal Makineler, tek bir fiziksel makinede birden fazla ve farklı işletim sisteminin çalışmasına izin vererek tam bir donanım sistemini taklit eder. Bir hipervizör tarafından desteklenen bu teknoloji, donanımı soyutlar ve her bir sanal makineye bir dizi özel kaynak ayırarak tam bir sanal işletim sistemi oluşturulmasını sağlar. Sanal makineler, çeşitli bilgi işlem ortamlarında yüksek derecede izolasyon ve uyumluluk sağlama konusunda mükemmeldir, bu da onları tam işletim sistemi kontrolü gerektiren uygulamalar için uygun hale getirir.

Buna karşılık Docker, konteynerleştirme teknolojisini kullanarak uygulama dağıtımında verimliliği ve hızı en üst düzeye çıkarmak için tasarlanmıştır. Sanal makinelerin aksine, Docker konteynerleri tam bir işletim sistemi oluşturmaz; bunun yerine, tek bir paylaşılan işletim sistemi çekirdeği üzerinde çalışan uygulamayı ve bağımlılıklarını içerir. Bu mimari fark, Docker'ın önyükleme süresini ve kaynak ek yükünü önemli ölçüde azaltan, aynı fiziksel sunucuda birden fazla uygulamayı veya mikroservisi yönetmek için yüksek ölçeklenebilirlik ve performansı destekleyen hafif, çevik konteynerleştirme yaklaşımının temelini oluşturur.

3.3.2.2. Kaynak Yönetimi

Sanal makinelerde kaynak tahsisi nispeten statiktir, her sanal makine sabit miktarda kaynak ayırır, bu da yetersiz kullanım veya kaynak kıtlığına yol açabilir. Docker ise daha dinamik bir kaynak tahsis modeli benimseyerek konteynerlerin

kaynakları talep üzerine kullanmasına olanak tanır. Bu esneklik, değişken iş yüklerine sahip uygulamalar için çok önemlidir, genel sistem verimliliğini artırır ve israfı azaltır.

3.3.2.3. Performans

Sanal makineler tam bir işletim sistemi çalıştırdığı için ek yük ve kaynak taleplerine neden olur. Docker ise üzerinde çalıştığı sistem kaynaklarını paylaşarak kullanır. Bu nedenler sanal makineler daha yavaştır. Docker konteynerleri verimlilik için ana bilgisayar çekirdeğinden yararlanarak daha hızlı başlatma süreleri ve daha düşük kaynak tüketimi sağlar.

3.3.2.4. Güvenlik

Sanal makineler, uygulamaları donanım düzeyinde izole ederek sağlam bir güvenlik çerçevesi sunar ve bu da onları sistemler arası güvenlik ihlallerine karşı daha güçlü hale getirir. Her sanal makine kendi güvenlik protokolleriyle bağımsız olarak çalışır, bu da onları yüksek güvenlik veya tam izolasyon gerektiren uygulamalar için uygun hale getirir.

Docker'ın güvenliği, konteynerlerin ana işletim sistemi çekirdeğini paylaşması nedeniyle bir şekilde tehlikeye girmektedir. Bir konteynerde bir güvenlik açığından yararlanılırsa, bu diğerlerini de etkileme potansiyeline sahiptir ve güvenliğin çok önemli olduğu ortamlarda önemli bir risk oluşturur. Buna rağmen, Docker bu tür riskleri azaltmak için gelişmiş güvenlik kontrollerini entegre etmiştir, ancak sanal makinelere kıyasla daha az izole kalmaktadır.

3.5. Kubernetes

Yazılım geliştirme alanında, verimli uygulama dağıtımı ve yönetimine duyulan ihtiyaç, konteyner orkestrasyon araçlarının ortaya çıkmasına neden olmuştur. Açık kaynaklı bir sistem olan Kubernetes bu teknolojilerin en önemlilerindendir (Phelan, 2021). Google tarafından tasarlanan ve sonrasında CNCF altında geliştirilmesi sürdürülen Kubernetes, konteynerizasyon araçları tarafından oluşturulan konteynerleri yönetir.

Kubernetes daha öncesinde Google tarafından geliştirilen ve kullanılan, bir cluster yönetim sistemi olan Borg yazılımından esinlenerek geliştirilmiştir (kubernetes, 2015). Kubernetes, Borg'un bazı sınırlamalarının üstesinden gelmeyi ve yeteneklerini daha geniş bir kullanıcı tabanına yaymayı amaçlayan Borg'un açık

kaynaklı bir sürümü olarak tasarlanmıştır. Geliştirilmesinde Go programlama dili kullanılmıştır.

Kubernetes'in kullanımı birçok avantajı beraberinde getirir. Kubernetes'teki otomatik ölçeklendirme, değişen iş yüklerini ele almada sistemin verimliliğini ve uyumunu artıran temel bir özelliktir. Bu özellik, Kubernetes'in bir uygulamanın çalışan örneklerinin sayısını mevcut talebe göre otomatik olarak ayarlamasına olanak tanır, böylece optimum kaynak kullanımı sağlar ve manuel müdahale olmadan performansı korur. Otomatik ölçeklendirme yalnızca kaynak verimliliğine katkıda bulunmakla kalmaz, aynı zamanda statik cluster kurulumlarında aşırı provizyon ihtiyacını azaltarak maliyet yönetimine de katkıda bulunur. Kubernetes, karmaşık uygulamaların ihtiyaçlarına gerçek zamanlı olarak uyum sağlayabilen esnek bir altyapıya sahiptir.

Kubernetes herhangi bir nedenle çalışmayan konteynerleri otomatik olarak değiştirir veya yeniden başlatır. Kendi kendini iyileştirme özelliği ve trafiğin yalnızca çalışan konteynerlere yönlendirilmesini sağlayan servis keşfi ve yük dengeleme yetenekleri vardır. Kubernetes çoklu hata toleransı seviyelerini destekler. Bileşenler arızalandığında bile uygulamaların sorunsuz bir şekilde çalışmaya devam etmesini sağlar. Bu, Kubernetes'i kritik uygulamalar için güvenilir bir platform haline getirir.

Kubernetes geri alma özellikleri sayesinde yeni bir dağıtımın sorunlara yol açması durumunda bir uygulamanın önceki sürümlerine kolayca geri dönülmesini sağlar. Bu, yeni özelliklerin ve güncellemelerin durumu bozmadan eklenebileceği ve test edilebileceği istikrarlı bir dağıtım sağlar.

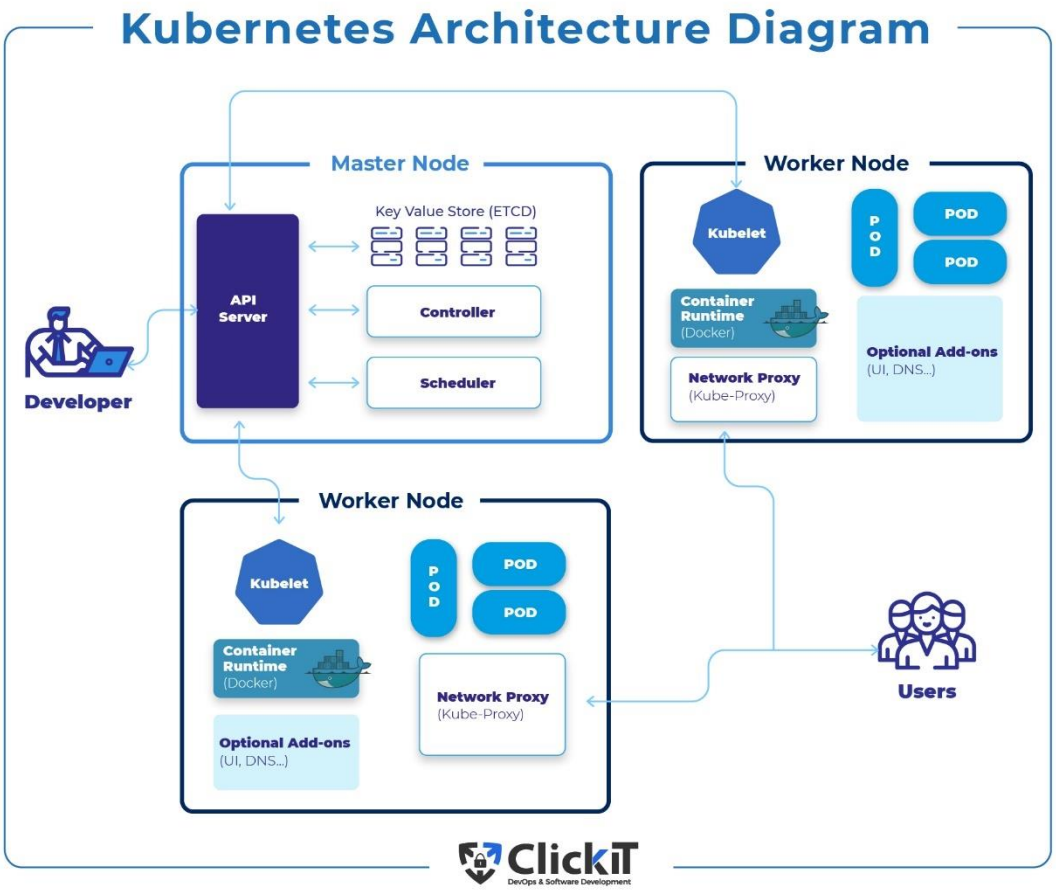
Kubernetes, çeşitli araçlar aracılığıyla sistemin metriklerini izleyebilen ve işleyişi hakkında içgörüler sağlayabilen ayrıntılı izleme yeteneğine sahiptir. Bu veriler performans ayarlama, kapasite planlama ve anomali tespiti için hayati önem taşır ve yöneticilerin ve geliştiricilerin yüksek hizmet kalitesi ve kullanılabilirlik düzeylerini korumalarını sağlar.

Kubernetes yetenekleri ile mikroservis mimarisini destekler. Bir mikroservis ekosisteminde çevik geliştirme ve dağıtım döngüleri için gerekli olan CI/CD uygulamalarını kolaylaştırır. Böylece geliştiricilerin ve kuruluşların bulut teknolojilerin tüm potansiyelinden yararlanmasına da olanak tanır.

3.5.1. Kubernetes'in Temel Bileşenleri

Kubernetes yapısı master ve worker node olmak üzere iki ana bileşenden oluşur. Bu bileşenlerde kendi içinde bölümlere ayrılmaktadır. Kubernetes'i oluşturan bütün parçalar bu bölüm başlığı altında incelenecektir.

Şekil 6: Kubernetes Bileşenleri



Kaynak: (Pantic, 2022)

Kubernetes mimarisini oluşturan bileşenleri anlamak için bilinmesi gereken kavramlar vardır. Bunlar aşağıda açıklanmıştır.

Cluster: Konteyner uygulamaları çalıştıran node kümesidir. Bir cluster en az bir master node ve birden fazla worker node'dan oluşur.

Pod: Bir veya birden fazla konteyneri içeren en küçük Kubernetes birimidir. Pod'lar, depolama ve ağ gibi kaynakları paylaşan bir veya daha fazla birbiriyle yakından ilişkili konteyneri çalıştırmak için tasarlanmıştır. Her bir Pod'a cluster içinde benzersiz IP

adresi tahsis edilir, bu da diğer Pod'larla ve harici servislerle iletişim kurmasını sağlar. Pod’lar geçici Kubernetes varlıklarıdır.

Service: Mantıksal bir Pod kümesini ve bunlara erişmek için kuralları tanımlayan bir soyutlamadır. Bu, yük dengeleme gibi ayarları içerebilir. Servisler, uygulamaların trafik almasını sağlar.

Volume: Veri tutan Pod'lara bağlı bir depolama birimidir. Bir Pod'un geçici yerel depolamasının aksine, bir Volume kalıcıdır ve Pod yeniden başlatıldığında hayatta kalır. Yapılandırılması Kubernetes API'si içinde yönetilir.

Namespace: Birden fazla ekibe veya projeye yayılmış çok sayıda kullanıcının bulunduğu ortamlarda kullanılmak üzere tasarlanmıştır. Cluster kaynaklarını birden fazla kullanıcı arasında bölmenin bir yoludur.

Deployment: Pod'lara ve bunların altında yatan ReplicaSet'lere güncellemeler sağlamak için tasarlanmıştır. Bir Deployment, Pod'lar ve ReplicaSet'ler için hangi kapsayıcı imajlarının kullanılacağı, çalıştırılacak replika sayısı ve güncelleme ve geri alma işlemlerinin nasıl yapılacağı gibi güncellemeleri tanımlamanıza olanak tanır.

Ingress: Bir cluster servislere, tipik olarak HTTP'ye harici erişimi yöneten bir API nesnesidir. Ingress yük dengeleme, SSL sonlandırma ve ad tabanlı sanal barındırma sağlayabilir. İstekleri, istek ana bilgisayarına veya yoluna dayalı olarak hizmetlere yönlendirmenin ve bu kuralları tek bir kaynakta merkezileştirmenin bir yoludur.

3.4.1.1.Master Node

API Server, Etcd, Controller ve Schedular bileşenlerinden oluşur.

API Server: Cluster’a yapılan tüm REST isteklerini alan merkezi yönetim bileşenidir. Kullanıcıların, yönetim araçlarının ve diğer bileşenlerin cluster ile etkileşime girmesine olanak tanır.

Etcd: Kaynakların yapılandırılmasını, hizmetlerin keşfedilmesini ve cluster gibi dağıtılmış sistemlerin koordinasyonunu kolaylaştıran açık kaynaklı, dağıtılmış bir anahtar/değer deposudur (Armo, 2024).

Controller: Birden fazla controller’dan meydana gelen master node bileşenidir. Controller’lar, görevi API server aracılığıyla cluster’ın mevcut durumunu izlemek ve mevcut durumu istenen duruma getirmeye çalışan değişiklikler yapmak olan bileşenlerdir.

Schedular: Kaynak kullanılabilirliğine ve diğer zamanlama kısıtlamalarına dayalı olarak Pod’lar biçimindeki işleri çalışan düğümlere atamaktan sorumludur. Mevcut iş

yükünü ve kaynak gereksinimlerini göz önünde bulundurarak bir Pod için en uygun düğümü seçer.

3.4.1.2.Worker Node

Konteynerlerin çalıştığı yerdir. Kubelet, Kube-Proxy, Container Runtime bileşenlerinden oluşur.

Kubelet: Her bir node'da çalışan bir ajandır. Konteynerlerin öngörüldüğü şekilde bir Pod içinde çalışmasını sağlamaktan sorumludur.

Kube-Proxy: Kubernetes'in Pod'lar ve servisler arasında ağ iletişimi sağlayan önemli bir bileşenidir (Adamson, 2023).

Container Runtime: Konteynerleri çalıştırmakla görevlidir. Konteynerlerin çalışması için gerekli altyapıyı sağlar.

ÜÇÜNCÜ BÖLÜM
TEKNOLOJİK ADAPTASYON YOLLARI

Mikroservis mimarisine geçiş sorunsuz bir şekilde yapılabilmesi için belirli yöntemler uygulanmalı ve adımlar izlenmelidir. Bu bölümde zamanla kazanılmış tecrübelerden yararlanılarak elde edilmiş yaklaşımlar ve geçiş adımları ele alınmaktadır.

1. GEÇİŞ STRATEJİLERİ

1.1. Strangler ve Big Bang Yaklaşımları

Yazılım mimarisi dönüşümünde monolitik bir mimariden mikroservis mimarisine geçişte, iki yaygın geçiş stratejisi kullanılmaktadır: Big Bang yaklaşımı ve Strangler yaklaşımı. Her bir strateji farklı metodolojiler, avantajlar ve zorluklar sunmaktadır.

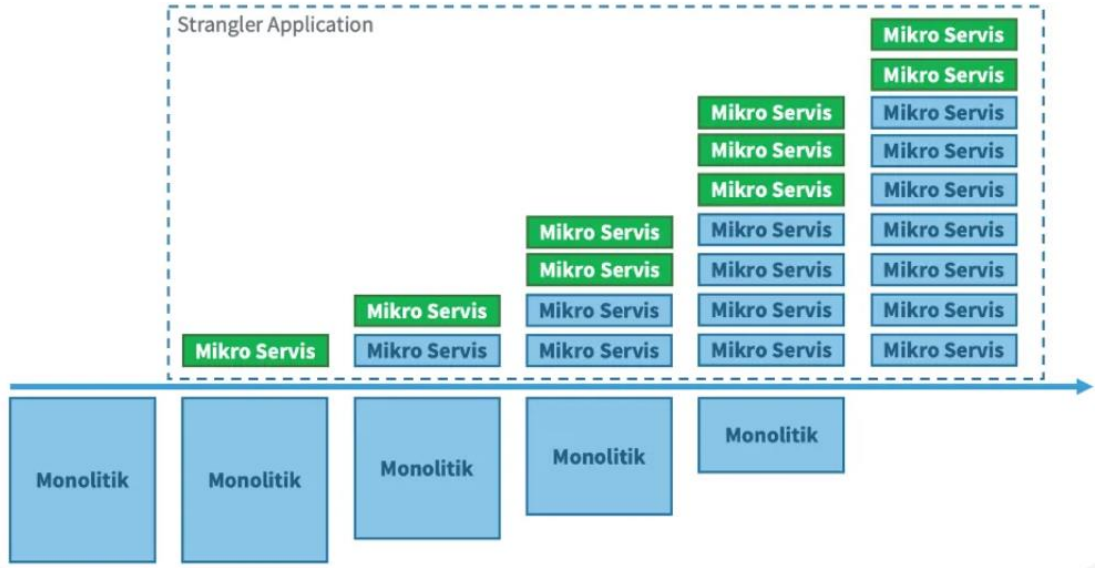
Big Bang geçiş yaklaşımı, tüm bileşenlerin monolitik mimariden mikroservis mimarisine tek seferde geçişini önerir. Bu strateji, tüm uygulamanın durdurulması, yeniden yapılandırılması ve mikroservis odaklı biçiminde tek bir ortak çabayla dağıtıma alınması demektir. Bu yaklaşımın ana avantajı, basit geçiş takvimi ve yeni bir mimari modele geçerek iki mimariyi paralel olarak çalıştırma aşamasını azaltmasında yatmaktadır. Ancak Big Bang yaklaşımı risklidir. Temel riskler, geçiş sırasında sistemin çökmesi ve sistem genelinde hataların ortaya çıkması olasılığıdır. Bu yaklaşımda birden geçiş yapılması nedeniyle, yeni sistemde öngörülemeyen herhangi bir sorun önemli aksaklıklara yol açabilir. Bu yöntem ayrıca, sistemin tüm parçalarının tam olarak anlaşılması ve yeni mimaride işlev görecek şekilde titizlikle yeniden tasarlanması gerektiğinden, başlangıçta kapsamlı bir planlama gerektirir.

Strangler yaklaşımı ise monolitik mimarinin tek tek bileşenlerinin veya servislerinin metodik ve sıralı bir şekilde mikroservis mimarisine dönüştürüldüğü kademeli bir geçişi önerir. Bu yaklaşımda monolitik yapı ile mikroservisler birlikte çalışır ve yeni istekler mikroservis olarak geliştirilir. Yazılım uygulamasının belirli bölümlerinin izole edilmesine, dönüştürülmesine ve bağımsız olarak dağıtılmasına olanak tanıyarak kesinti en aza indirilir ve sürekli test ve konfigürasyon yapılması sağlanır.

Strangler yaklaşımı, belirli bir zamanda sistemin yalnızca bazı bölümleri değiştirildiğinden, yaygın sistem arızası riskinin azaltılması da dahil olmak üzere çeşitli avantajlar sunar. Ayrıca bu yöntem, daha esnek bir geliştirme ortamını destekler ve çevik geliştirme uygulamalarıyla daha uyumludur. Bununla birlikte Strangler yaklaşımı toplam geçiş süresini uzatır ve orijinal monolitik sistemin kalan parçalarının yeni mikroservislerle sürekli olarak etkileşime girmesi gerektiğinden karmaşıklığı geçici olarak artırır. Bu, sağlam bir ara entegrasyon mekanizması gerektirir ve geçiş döneminde iki mimari tarzın sürdürülmesinde zorluklara yol açabilir.

Özetle, Big Bang ve Strangler yaklaşımları arasındaki seçim uygulamanın özel ihtiyaçlarına, risk toleransına ve operasyonel yeteneklerine bağlıdır. Big Bang yaklaşımı hızlı ancak riskli bir geçiş sunarken, Strangler yaklaşımı daha kontrollü, ancak uzun süreli bir geçiş yolu sağlar ve tipik olarak sürekli teslimat ve modern çevik metodolojilerle daha iyi uyum sağlar.

Şekil 7: Strangler Yaklaşımı



Kaynak: (Çilingir, 2023)

1.2. Pilot Projeler

Kurumda mikroservis mimarisi ile projeler geliştirmeye ve yönetmeye başlamanın en ideal şekilde yapılabilmesi için pilot projeler yapmak en uygun metotlardan biridir. Monolitik bir mimariden mikroservis mimarisine geçişin ilk

aşaması için pilot projeler önerirken, birkaç stratejik kriterler göz önünde bulundurulmalıdır. Bu kriterler riskleri en aza indirmeyi, öğrenme fırsatlarını en üst düzeye çıkarmayı ve mikroservis mimarisinin faydalarını etkili bir şekilde göstermeyi amaçlamaktadır. Pilot projelerin seçilmesi ve tasarlanmasına ilişkin kritik hususlar aşağıda belirtilmiştir:

- ✓ Yazılım uygulamalarının olası arızalarının etkisini en aza indirmek için kritik olmayan uygulamalar seçmek gerekir. Bu uygulamalar geliştiricilere geçişin somut faydalarını gösterebilmelidir.
- ✓ Monolitik mimari içinde nispeten diğer bileşenlere göre daha az bağımlılığı olan bileşenler seçilmelidir. Böylece yazılım uygulamasının diğer kısımlarındaki değişiklik ihtiyacı azalır ve geçişin karmaşıklığı en aza iner.
- ✓ Performans sorunları olan bileşenlere öncelik verilmelidir. İlk olarak bu bileşenlerin dönüştürülmesi performans iyileştirmeleri sağlar.
- ✓ Başarı için net ölçütler sağlayabilen bileşenler tercih edilmelidir. Performans, dağıtım döngüleri ve kaynak kullanımındaki iyileştirmelerin kolayca ölçülebildiği ve gözlemlenebildiği hizmetleri seçmek avantajlıdır.
- ✓ Geliştirme ve operasyon ekiplerinin mevcut beceri ve uzmanlıklarının olduğu teknolojiler ve uygulamalar seçilmelidir.
- ✓ Daha az düzeltmeye ihtiyaç olan bileşenlerle başlamak avantajlıdır.
- ✓ Pilot proje sırasında ve sonrasında geri bildirim önemlidir. Bu, etkiyi değerlendirmek ve iyileştirmek için izleme araçlarını ve kullanıcı geri bildirim kanallarını içerir.

Kurum, pilot projeleri bu kriterlere göre dikkatlice seçerek mikroservis mimarisine geçişle ilgili riskleri etkili bir şekilde yönetebilir, faydaları erkenden gösterebilir ve sonraki geçişler için bir örnek oluşturabilir.

2. GEÇİŞ ADIMLARI

Monolitik mimari ile oluşturulmuş bir yapıdan mikroservis mimarisine geçişte belli bir algoritmayı takip etmek gerekir. Bu geçişte farklı yöntemler ve teknolojiler kullanılabilse de algoritmanın ana hatları aynıdır.

2.1. Monolitik Uygulamayı Analiz Etme

Monolitik bir mimariden mikroservis mimariye geçişin ilk aşaması, mevcut monolitik sistemin titiz bir şekilde incelenmesini ve ayrıştırılmasını gerektirir. Bu süreç, geçişte en etkili stratejileri belirlemek için gerekli olan sistemin bileşenlerini ve bağımlılıklarını tespit edilmesini de kapsar.

Monolitik mimarinin kapsamlı bir şekilde anlaşılması için eğer mevcut ise dokümantasyonun kapsamlı bir incelemesi yapılır. Bu inceleme, dokümantasyonun eksik veya eski olduğu, sistemin mevcut durumunu doğru bir şekilde yansıtmak için güncelleme veya tamamen yeniden yazma gerektiren alanların belirlenmesine yardımcı olur.

Mevcut performans verileri analiz edilerek sistemin verimliliğini ve yanıt verebilirliğini ölçülebilir. Bu kıyaslama, mikroservis mimarisine geçişin genel sistem performansı üzerindeki etkisini değerlendirmek için geçiş sonrası karşılaştırmalı bir analiz yapılmasına olanak tanıyan bir referans noktası olarak kullanılabilir.

Monolitik sistemin incelenmesiyle temel işlevler ve modüller belirlenir. Amaç, bu mikroservislerin bağımlılığını azaltmak, bakım kolaylığı, ölçeklenebilirlik ve bağımsız dağıtımı sağlayabilecek şekilde ayırmaktır.

Uygulamanın teknik borcunun bir değerlendirmesi yapılır. Bu, geçiş sürecini engelleyebilecek eski kodların, eski teknolojilerin ve ölçeklenemeyen bileşenlerin değerlendirilmesini sağlar. Teknik borcun ele alınması, mikroservis mimarisine geçişin verimsizlikleri veya potansiyel başarısızlık noktalarını sürdürmemesini sağlamak için çok önemlidir.

Özetle, monolitik sistemin analizi, mikroservis mimarisine geçişin temel bir adımını oluşturur. Analiz mikroservis mimarisine yönelik geçiş için zemin hazırlar. İş hedefleri ve teknik gereksinimlerle uyumu sağlar.

2.2. Bileşenleri Tanımlama

Monolitik mimariden mikroservis mimarisine geçişin bu adımında ana odak noktası, mikroservis mimarisine dönüştürülebilecek bileşenleri tanımlamaktır. Monolitik mimarilerde kodlar zamanla karmaşıklaşır ve iç içe geçmeye başlar. Aynı

işlevleri yerine getiren benzer fonksiyonlar oluşur. DDD ilkelerinin kullanılması bu süreçte düzgün bir ayrım yapılmasına olanak tanır. Monolitik sistem içindeki sınırlı bağlamlar (belirli iş sorumluluklarına ve iyi tanımlanmış arayüzlere sahip farklı işlevsel bölümler) tanımlanarak mikroservis mimarisi için mantıksal bölümler belirlenir. Bu sınırlı bağlamlar, benzersiz iş yeteneklerine göre tanımlanmış olup, iş alanlarının net bir şekilde ayrılmasını sağlar.

Mikroservislerin uygun sınırlarının belirlenmesi bu adımdaki en önemli zorluktur. Belirlenen her bir servis, tek bir işlevi yerine getirecek veya birbiriyle yakından ilişkili bir dizi işlevi yerine getirecek kadar küçük olmalı ve böylece yüksek uyumluluk sağlanmalıdır. Bu DDD ilkeleriyle yapılırsa geliştirme, test ve bakım kolaylığı sağlanmış olur.

Mikroservisler için uygun bileşenlerin belirlenmesinin yinelemeli bir şekilde yapılması daha yararlı olabilir. Bu yinelemeli yaklaşım geçiş süreci sırasında hata riskini azaltır.

Mikroservislere dönüştürülmek üzere monolitik sistemdeki bileşenlerin belirlenmesi, dikkatli planlama, alan uzmanlığı ve yinelemeli iyileştirme gerektiren kritik bir aşamadır.

2.3. Mikroservisleri Tanımlama ve Geliştirme

Önceden tanımlanmış bileşenlerin mikroservis olarak geliştirilmesini içerir. Mikroservisler, sistemdeki diğer servislerden bağımsız olarak geliştirilmeye olanak tanıyan kendi teknoloji yığınları ve veritabanları ile özerk bir haldedir.

Mikroservislerin geliştirilmesi, ayrı dağıtım, ölçeklendirme ve testlere izin vermek için her mikroservise ayrı geliştirme ortamlarının kurulmasını gerektirir. Bu ayrım, geliştirme sürecinin çevikliğini artırarak ekiplerin sistemin diğer kısımlarını etkilemeden güncellemeleri ve iyileştirmeleri hızla uygulamasına olanak tanır. Geliştirme sırasında, hizmetlerin gevşek bağlı bir şekilde entegre edilmesi için gerekli olan servisler arası iletişim API'ler ile sağlanmalıdır. Bu sebeple uygulama için gerekli API’ler yazılmalıdır.

Her mikroservis, diğer mikroservislerle paylaşılmayan veritabanını kontrol eder ve böylece veri kapsüllemesini sağlar. Bu bağımsızlık, her bir servisin yönettiği verilerin bütünlüğünü ve güvenliğini korumak için önemlidir.

2.4. Mikroservisler Arası İletişim

En uygun iletişim tarzını (senkron veya asenkron) seçmek için mikroservisler arasındaki veri alışverişi gereksinimleri değerlendirilmelidir. HTTP gibi senkron iletişimler gerçek zamanlı istemci-sunucu etkileşimleri için uygunken, mesaj kuyrukları veya olay akışları kullanan asenkron iletişimler ayrıştırılmış, olay odaklı mimarileri destekler. Seçilen iletişim tarzına bağlı olarak, hizmetlerin operasyonel ve performans ihtiyaçlarına en uygun protokoller seçilmelidir.

Eşzamanlı iletişim için ölçeklenebilir ve kullanımı kolay, durum bilgisi olmayan REST API'ler kullanılmalıdır. Bu API'lerin iyi belgelendirilmeli ve mevcut hizmet işlemlerini kesintiye uğratmadan değişiklikleri yapmak için sürümlendirilmelidir.

Yüksek trafik veya hizmet kesintisi sırasında bile güvenilir mesaj teslimatı sağlamak için mesaj kuyrukları kullanılabilir. Mesaj kuyrukları, başarılı bir şekilde işlenene kadar mesajları arabelleğe alır, böylece veri kaybını önler ve yük seviyelendirmeyi sağlar. Mesajların alındığını ve işlendiğini onaylamak için sistemler kurulabilir. Başlangıçta işlenemeyen mesajlar için yeniden deneme mekanizmaları uygulayarak geçici arızalar nedeniyle hiçbir mesajın kaybolmaması sağlanır.

Verimi ve gecikme süresini etkin bir şekilde yönetmek için iletişim kanallarının performansı sürekli olarak izlenmelidir. İletişim altyapısındaki darboğazları veya arızaları tespit etmek için izleme araçları kullanılmalıdır. İstemci isteklerini birden fazla mikroservise verimli bir şekilde dağıtarak kaynak kullanımını ve yanıt süresini optimize etmek için yük dengeleyicileri kullanılmalıdır.

2.5. Bağımsız Dağıtma

Bu adım, mikroservislerin özerk olarak çalışabilecekleri ve sistemdeki diğer servislerden ayrı olarak yönetilebilecekleri bir sunucuda fiilen çalıştırılmasıdır. Dağıtım yapılmadan önce, her bir mikroservisin kendi veritabanına, tanımlanmış API'lere ve izole edilmiş çalışma zamanı ortamına sahip olacak şekilde tamamen kapsüllenmesi sağlanmalıdır. Bu kapsülleme, mikroservisin sistemin diğer

bileşenleriyle olumsuz etkileşime girmeden bağımsız olarak çalışabilmesini garanti eder. Dağıtım süreci, geliştirme aşamasından üretime kadar tutarlı bir ortam sağlayan ve ortamlar arasındaki tutarsızlık olasılığını azaltan Docker gibi konteynerleştirme teknolojilerini içermelidir.

Servis keşfi, yük dengeleme ve otomatik ölçeklendirme dahil olmak üzere dağıtım karmaşıklıklarını yönetmek için Kubernetes gibi hizmet düzenleme araçları sürece dahil edilmelidir.

Geliştirme, test ve dağıtım süreçlerini otomatikleştiren sağlam CI/CD süreçleri oluşturulmalıdır. Bu otomasyon, entegrasyon ve otomatik testler yoluyla sorunların erkenden yakalanmasına yardımcı olur. Tüm testleri geçen herhangi bir değişikliğin otomatik olarak üretime dağıtılabilmesi sağlanmalı ve manuel müdahale en aza indirilmelidir.

2.6. Veri Yönetimini Ayırma

Mikroservis mimarisinde bileşenleri ayrıştırma hem uygulama hem veritabanı bazında işlemler gerektiren bir adımdır. Monolitik mimaride tek bir veritabanı içinde bulunan veriler sistemin doğası gereği mikroservis mimarisinde ayrı olarak tutulmalıdır.

Mikroservis mimarisinde veri sahipliği ilkesi, her bir mikroservisin kendi verileri üzerinde özel kontrole sahip olmasını zorunlu kılar. Bu, hizmetlerin gevşek bir şekilde bağlanmasını sağlayarak özerkliği destekler ve sistem esnekliğini artırır. Bu bilgiler dikkate alındığında ideal bir yapı kurmak için veritabanı ayrıştırılmalıdır.

Mikroservisler arasında veri dolaşımını önlemek için her mikroservis kendi veritabanına sahip olmalıdır. Bu izolasyon, sistemin diğer bölümleri yoğunluk altındayken veya bakımdan geçerken bile mikroservisin verilerinin bütünlüğünü ve tutarlılığını korumaya yardımcı olur. Veritabanları her bir mikroservisin ihtiyaçları için özel olarak ayarlanmalıdır. Bu yaklaşım, veri modelinin karmaşıklığı artıran ve performansı düşüren gereksiz veri alanları olmadan mikroservisin işlevselliğini en iyi şekilde desteklemesini sağlar.

Uygulama mantığı ile veri erişim mantığı arasında net bir soyutlama katmanı sağlamak mikroservisleri veritabanına daha az bağımlı hale getirir. Her bir mikroservisin ihtiyaçlarına özel DAO geliştirerek ayrıştırılmış veri etkileşimleri sağlanmalıdır.

Değişiklikleri sistematik bir şekilde yönetmek ve zaman içindeki veri değişikliklerini anlamak ve sorun gidermek için çok önemli olan veritabanı evriminin geçmişini korumak için veritabanı şemalarına sürüm kontrolü uygulanmalıdır. Değişiklikleri kontrollü ve geri döndürülebilir bir şekilde uygulamak için otomatik geçiş araçları kullanılmalıdır.

2.7. Optimize Etme ve Ölçeklendirme

Performans darboğazlarını belirlemek ve çözmek için her bir mikroservis sürekli olarak izlenmelidir. Verimliliği ve yanıt sürelerini artırmak için veri erişim modelleri iyileştirilmeli ve servis mantığı geliştirilmelidir.

Kullanım istatistiklerine ve performans ölçümlerine göre kaynak tahsisleri ayarlanmalıdır. Ayrıca herhangi bir hizmetin sistem kaynaklarını tekeline almasını önlemek için kaynak kotaları ve sınırları uygulanmalıdır.

Gerçek zamanlı kullanıcı etkileşimleri, sistem günlükleri ve hata raporları dahil olmak üzere çeşitli kaynaklardan veri toplayan ve analiz eden sağlam geri bildirim mekanizmaları oluşturulmalıdır. Bu geri bildirim, hizmetlerin iyileştirilebileceği veya ayarlama yapılması gereken alanları belirlemek için önemlidir.

Mikroservis güncellemelerinde yinelemeli bir yaklaşım gereklidir. Burada en son içgörülere ve geri bildirimlere dayalı olarak düzenli olarak artımlı değişiklikler yapılır. Bu yaklaşım, kullanıcı ihtiyaçlarını ve sistem performansı hedeflerini daha iyi karşılamak için hizmetlerin sürekli gelişimini ve uyarlanmasını sağlar.

2.8. Monolitik Parçaları Çıkarma

Monolitik bir mimariden mikroservis tabanlı bir mimariye geçerken monolitik uygulamanın mikroservislere başarıyla taşınan parçalarının sistematik olarak kaldırılması gerekir. Bir mikroservis tamamen çalışır hale geldiğinde ve belirlenen tüm görevleri beklendiği gibi yerine getirdiği doğrulandığında, monolitik uygulama içindeki ilgili bileşen kullanımdan kaldırılmak üzere ayrılabilir. Hangi bileşenlerin

taşındığını ve hangilerinin kaldırılmayı beklediğini takip etmek için bu süreç boyunca detaylı belgeler tutulabilir.

Tamamen silmeden önce, monolitik bileşenleri tamamen kaldırmak yerine geçici olarak devre dışı bırakmak tavsiye edilir. Bu yaklaşım, sistemin kararlılığının izlenmesine ve mikroservisin devre dışı bırakılan parçanın işlevselliğini herhangi bir veri bütünlüğü kaybı olmadan yeterince karşıladığından emin olmasına olanak tanır. Bu gözlem süresi boyunca, mikroservis uygulamasındaki beklenmedik davranışları veya eksiklikleri tespit etmek için tüm sistem etkileşimleri titizlikle kaydedilmelidir. Önceden tanımlanmış bir gözlem penceresi boyunca herhangi bir sorun ortaya çıkmazsa, mikroservise duyulan güven teyit edilmiş olur.

İşlemler en az kritik bileşenlerden başlayarak ve sistemin daha önemli parçalarına doğru ilerleyerek aşamalı olarak yapılmalıdır. Her çıkarma işleminin ardından, istenmeyen sonuçların kalan monolitik bileşenleri veya sistemin genel işlevselliğini etkilememesini sağlamak için bir dizi test yapılmalıdır. Her kaldırma işleminden sonra yapılan kapsamlı testler, sistemin sağlam ve güvenilir kalmasını sağlamak için regresyon, performans ve entegrasyon testlerini içerir.

DÖRDÜNCÜ BÖLÜM
SOSYAL GÜVENLİK KURUMU İÇİN DEĞERLENDİRME

Gelişen dijital dönüşüm ortamında, bütün kamu kurumları gibi SGK de teknolojik adaptasyon gerektiren zorlukların üstesinden gelmek durumundadır. Teknolojik adaptasyonun sağlanabilmesi için sadece teknik fizibilite değil aynı zamanda kurumun personel kaynağının bu geçiş için stratejik uyumu da değerlendirilmelidir. Bu bölümde SGK'de mikroservis mimarisinin benimsenmesinin stratejik sonuçları araştırılmakta ve bu değişimin gelişmiş hizmet sunumu, operasyonel çeviklik vb. daha geniş hedeflerle nasıl uyumlu olduğu değerlendirilmektedir. Ayrıca mikroservis mimarisinin benimsenmesinin kurumun güvenli ve verimli sosyal güvenlik hizmeti sunma misyonuna olumlu katkıda bulunmasını sağlamak için kültürel ve teknik değişimler üstünde durulmaktadır.

1. STRATEJİK UYGUNLUK VE FİZİBİLİTE

1.1. Kurumsal Hedeflerle Uyumun Değerlendirilmesi

SGK için önemli bir kurumsal hedef, hizmetlerin tüm vatandaşlara erişilebilirliğini artırmaktır. Günümüzde erişilebilirliğin sağlanmasının önemli bir yolu da yazılım hizmetlerinin erişilebilirliğiyle olmaktadır. Direkt vatandaş tarafından kullanılan veya vatandaşa hizmet veren SGK personelinin kullandığı yazılım servisleri bu erişilebilirliğin bir parçasıdır. Mikroservis mimarisi, tüm sistemi etkilemeden güncellenebilen, iyileştirilebilen ölçeklendirilebilen ve bağımsız olarak dağıtılabilir hizmetlerin geliştirilmesini sağlayarak bu konuda önemli bir rol oynayabilir. Bu da yazılım servislerinin daha hızlı güncellenmesi ve hizmetlere daha kısa zamanda erişim sağlayabilir.

SGK'nin yeni mevzuat veya politika değişikliklerine yanıt olarak yazılımlarını hızlı bir şekilde güncellemesi gerekir. Mikroservis mimarisi, yazılım sistemini bölümlendirerek sistemin belirli bölümlerinin bağımsız olarak güncellenmesine olanak tanır. Sık değişen mevzuatın ve buna paralel olarak geliştirilen yazılım uygulamalarının kesintisiz ve hızlı entegrasyonunu sağlar. Yazılım sisteminin tümünün güncellemeden bölüm bölüm güncellenebilmesi sık değişen bir mevzuatı olan bir kurum için önemlidir.

Mikroservis mimarisinin erişilebilirliği artırma ve yenilikçilik gibi stratejik hedefler ile uyumlu olması, SGK için faydalı bir teknolojik strateji olma potansiyelini ortaya koymaktadır.

1.2. Teknik Fizibilite Çalışması

Bu çalışma kurumun mevcut yazılım altyapısının mikroservislere geçiş için ne kadar uygun olduğunu değerlendirmeyi amaçlamaktadır. Bu çalışmada, mevcut durum, ihtiyaçlar ve riskler ele alınacaktır. Ayrıca kurumun bu mimariyi benimseme sürecinde karşılaşabileceği teknik gereksinimler ve uyumluluk sorunları değerlendirilecektir.

Yapılması düşünülen tez çalışmasının çeşitli zorlukları mevcuttur. Mikroservis mimarisi büyük monolitik uygulamaları dönüştürmek için kullanıldığında avantaj getirmektedir. Büyük bir uygulamanın doğru bir şekilde bölünmesi ve tamamen mikroservislerden oluşan bir sisteme dönüştürülmesi uzun bir zaman gerektirir. Mevcut durumda SGK'de mikroservis mimarisine geçilmesi için neler yapılması gerektiğine odaklanacak ve olası bir geçişin aşamalı bir şekilde nasıl gerçekleştirileceği ele alınacaktır. Monolitik bir sistemden mikroservis mimarisine geçişin ilk adımı gösterilecektir.

1.2.1. Mevcut Durum

SGK'nin mevcut altyapısında halihazırda 620 adet uygulama hizmet vermektedir (Sosyal Güvenlik Kurumu, 2024). Bu uygulamaları çalıştırmak için gerekli olan donanım, işlemci gücü, ağ kapasitesi bulunmaktadır. Dolayısıyla mikroservis mimarisi ile oluşturulmuş bir uygulamanın da çalışabilmesi için gerekli donanım, işlemci gücü ve ağ hızı mevcuttur. Ayrıca yazılım uygulamalarını konteyner haline getirebilmek için Docker, konteynerlerin orkestrasyonu için Kubernetes, kaynak kod ve proje yönetimi için GitLab teknolojileri kullanılmaktadır. GitLab, CI/CD yaklaşımını desteklemektedir.

SGK, diğer devlet kurumları gibi verilerini ve yazılımlarını kendi sunucularında barındırmaktadır. Günümüzde şirketler için bir seçenek olarak değerlendirilebilecek bulut bilişim hizmeti satın alarak verilerini ve/veya yazılım uygulamalarını bulutta işlemek güvenlik ve veri gizliliği gibi endişeler sebebiyle yasal olarak mümkün olmamaktadır.

Mikroservis mimarisinin SGK için düzgün bir şekilde değerlendirilebilmesi ve sisteme neler katabileceğinin gösterilebilmesi için mevcut bir sistem olan Emektar4B uygulaması üzerinden ele almanın daha iyi bir yaklaşım olacağı düşünülmektedir. Emektar4B, 4/1-(b) kapsamındaki kişilerin aylık tahsis işlemlerinin yapıldığı web tabanlı bir uygulamadır. Emektar4B, 2011 yılında yayımlanmıştır ve bu zamana kadar geliştirilmesi sürdürülmektedir.

Emektar4B uygulamasının 2011 yılından bugüne kadar kod karmaşıklığı sürekli artmış ve hala artmaktadır. Farklı geliştiriciler tarafından yapılan değişiklikler, uygulama kodlarının anlaşılmasını ve uygulamanın bakımını zorlaştırmıştır. Ek olarak uygulamanın eski bir teknoloji ile geliştirilmesi doküman ve örnek eksikliğine yol açmaktadır. Bu da yeni gelen ve bu teknolojiye aşina olmayan geliştirici için büyük bir zorluk ortaya çıkarmaktadır. Ayrıca entegre edilmesi düşünülen yeni teknolojiler uyumluluk sorunları nedeniyle kullanılamamaktadır.

Emektar4B uygulaması monolitik yapıda geliştirildiğinden ve buna bağlı olarak zamanla yazılım bileşenleri arasındaki bağımlılık artması nedeniyle tek bir yerinde yapılan düzenleme veya ekleme bütün uygulamanın yeniden derlenmesini gerektirir. Bu da uygulamanın yeniden dağıtımı yapılırken kesintilere neden olmaktadır. Ayrıca kodlar arasındaki bağımlılık artmıştır. Bunun sonucu olarak kodun bir yerinde yapılan değişiklik kodun birçok yerinde değişiklik yapılmasını gerektirmektedir.

Mikroservis mimarisinin getirdiği avantajlardan faydalanabilmek için DevOps yaklaşımının benimsenmesi bir zorunluluktur. Mikroservisler dağıtık ve karmaşık yapısı nedeniyle yönetilmesi daha fazla çaba gerektiren ve işlerin operasyonel tarafına ağırlık getiren bir mimaridir. SGK'de mevcut durumda DevOps altyapısını işletebilecek yeterli personel bulunmamaktadır.

1.2.2. İhtiyaçlar ve Analiz

Kurumda mikroservis mimarisinin başarılı ve verimli bir şekilde uygulanabilmesi için çeşitli gerekliliklerin sağlanması gerekmektedir. Bunlar; personelin eğitilmesi, DevOps yaklaşımının benimsenmesi ve araçlarının kullanılması, mikroservisleri doğru bir şekilde bölebilmek için DDD gibi metotların bilinmesi, dağıtık veritabanı yönetiminin yapılabilmesi, mikroservis mimarisine uygun test

stratejilerinin benimsenmesi, mikroservisleri izleme araçlarının kullanılması olarak sıralanabilir.

Uygulama geliştirmeye yönelik dağıtık ve parçalı yapıya sahip mikroservis mimarisi, operasyonel karmaşıklıkları ve ölçeklenebilirlik ihtiyaçlarını karşılamak için doğası gereği DevOps uygulamalarının entegrasyonunu gerektirir. Bu birliktelik, mikroservis ortamında çeşitli nedenlerden dolayı büyük öneme sahiptir.

Mikroservis mimarisindeki dağıtım karmaşıklığı, her servis bağımsız olarak geliştirildikçe, dağıtıldıkça ve ölçeklendirildikçe önemli ölçüde artmaktadır. Geleneksel IT operasyonları, DevOps'un temel unsurları olan otomasyon ve standartlaştırılmış süreçler olmadan, birden fazla servisteki sık dağıtımları yönetmekte zorluk çekmektedir. DevOps, derleme, test ve sürüm döngülerinin otomatikleştirildiği ve hızlandırıldığı CI/CD uygulamalarına olanak tanır. Bu yalnızca dağıtımların hızını artırmakla kalmaz, aynı zamanda manuel işlemlerle ilişkili hata riskini de azaltır.

Ölçeklenebilirlik mikroservislerin temel bir avantajıdır; ancak çeşitli ortamlarda çok sayıda hizmet örneğinin yönetilmesinde zorluklara neden olur. DevOps uygulamaları, Kubernetes gibi konteyner orkestrasyon teknolojilerini içerirler. Bunlar mikroservislerin dinamik ölçeklendirilmesini ve yönetimini kolaylaştırır. Konteyner orkestrasyonu, mikroservis tabanlı mimarilerin yönetimi için çok önemli olan konteynerli uygulamaların dağıtımını, ölçeklendirilmesini ve çalışmasını otomatikleştirir.

Dağıtılmış bir sistemde izleme ve loglama karmaşıktır ancak sistem sağlığını ve performansını korumak için hayati öneme sahiptir. DevOps, mikroservislerin bütünsel bir görünümünü sağlayan, etkileşimlerini ve performanslarını gerçek zamanlı olarak izleyen entegre izleme araçlarını içerir. Bu tür araçlar, hataların hızlı bir şekilde belirlenmesine ve çözülmesine yardımcı olur; bu, servislerin gevşek bir şekilde bağlı olduğu ve ağa bağımlı olduğu bir sistemde çok önemlidir. Kurumda halihazırda izleme ve hataları tespit etmeye yardımcı araçlar kullanılmamaktadır. Bu araçların temin edilerek kurumda kullanıma sunulması gerekmektedir. Ayrıca kurum personelinin de DevOps yaklaşımını benimsemesi gerekmektedir.

1.2.4. Riskler

Mikroservis mimarisine geçiş ölçeklenebilirlik, esneklik ve dağıtım hızı gibi çok sayıda avantaj sunarken aynı zamanda çeşitli riskleri ve zorlukları da beraberinde getirir. Bu riskleri anlamak, kurumlar için kritik öneme sahiptir.

Mikroservisler sistem karmaşıklığını artırır. Düzinelerce veya yüzlerce ayrı servisi yönetmek, tek bir monolitik uygulamayı yönetmekten daha zor olabilir. Her servisin kendi bağımlılıkları, kaynak gereksinimleri ve ölçeklendirme politikaları olabilir; bu da dağıtım, izleme ve yönetimi zorlaştırabilir.

Veri yönetimi, mimarinin dağıtık yapısı nedeniyle mikrosevislerde daha karmaşık hale gelir. Monolitik bir veritabanı sisteminin geleneksel ACID özellikleri olmadan hizmetler arasında veri tutarlılığının sağlanması, olay kaynağı veya dağıtılmış işlemler gibi stratejilerin uygulanmasını gerektirir ve bu da karmaşıklığı artırabilir.

Mikroservisler sıklıkla bir ağ üzerinden iletişim kurar ve bu da gecikmeye neden olur. Bu durum, özellikle servisler kötü tasarlanmışsa veya ağ altyapısı yeterince sağlam değilse uygulama performansını düşürebilir. Ayrıca, ağ sorunları servisin kullanılamamasına veya servisten servise iletişimde arızalara yol açabilir.

Mikroservisler kaynak kullanımının artmasına da neden olabilirler. Her mikroservis ayrı çalışma zamanı ortamları veya veritabanları gerektirebilir. Bu monolitik bir mimariye kıyasla potansiyel olarak daha yüksek operasyonel maliyetlere yol açabilir.

Mikroservislerde güvenlik daha karmaşıktır. Her servis, güvenlik ihlalleri için potansiyel bir giriş noktasıdır. Erişim kontrollerini yönetmek ve bağlantıları şifrelemek de dahil olmak üzere tüm servislerde tutarlı bir şekilde güvenlik uygulaması daha zordur.

Test işlemleri monolitik uygulamalara kıyasla daha zordur. Mikroservislerin hem ayrı ayrı hem de diğer mikroservislerle birlikte test edilmesi gerekmektedir. Ayrıca uçtan uca testler için gerçek ortam koşullarını simüle etmek zor olabilir.

Sürekli dağıtım mikroservislerin ana avantajlarından biridir fakat düzgün bir şekilde yönetilmezse risk oluşturabilir. Dağıtım sıklığı, uygun CI/CD düzeni mevcut değilse istikrarsızlık sorunlarına yol açabilir.

Kurumdaki yazılım ekiplerinin birbiriyle koordinasyonu sağlaması gerekliliği mikroservis mimarisine uyumsuzluk riskini doğurabilir. Çünkü Mikroservis mimarisi kültürel bir değişimin sağlanmasını gerekli kılar.

2. ANKET UYGULAMASI

2.1. Anketin Amacı

Araştırmanın amacı, görece yeni bir yazılım mimarisi olan mikroservis mimarisine olası adaptasyon için BT personelinin mevcut becerilerini, bilgilerini ve hazır olma durumlarını değerlendirmektir. Deneyim, kullanılan teknolojik araçlar ve hazır olma durumları değerlendirilerek iş gücü ve altyapıdaki güçlü ve zayıf yönlerin belirlenmesi amaçlanmaktadır. Bu yönler, monolitik mimariden mikroservis mimarisine geçiş için planlama ve karar alma birimlerinin bilgilendirilmesine yarayabilir. Gerekli eğitim, kaynak tahsisi ve değişim yönetimi süreçlerinin etkili bir şekilde ele alınabilmesine imkân tanır.

2.2. Anketin Kapsamı

Araştırma, Hizmet Sunumu Genel Müdürlüğünde çalışan bilişim uzmanı, çözümleyici, mühendis, programcı, sosyal güvenlik uzmanı ve sosyal güvenlik uzman yardımcısı kadrolarında bulunan personellere odaklanılmıştır. 62 personel ankete katılmıştır. Bu sayı bu kadrolarda çalışan personel sayısının %19'una tekabül etmektedir.

2.3. Anket

Araştırmada verileri elde etmek için anket formu kullanılmıştır. Anket formu online bir anket portalı olan SurveyMonkey sitesinde hazırlanmıştır. Anket 10 sorudan oluşmaktadır. İlk 2 soru katılımcıların demografik bilgilerinin istendiği çoktan seçmeli sorulardır. 3., 4. ve 5. sorularda katılımcıların becerilerinin mimari geçiş için uygunluğu ve tecrübelerinin yeterliliği ölçülmek istenmiştir. 3. soru çoklu cevap seçeneği sunularak becerilerin kapsamı öğrenilmesi amaçlanmıştır. 4. ve 5. soru çoktan seçmeli sorular olup tecrübe bilgisine odaklanılmıştır. Çünkü pratik tecrübe yazılım alanında en çok ihtiyaç duyulan olgulardan biridir. 6. soruda personelin kullandığı yazılım araçları sorulmuş ve çoklu cevap seçeneğine olanak tanınmıştır.

Seçenek olarak sunulan yazılım araçları mikroservis mimarisi ile birlikte en fazla kullanılan araçlardır. Soruya diğer seçeneği eklenerek eklenmek istenilen yazılım araçlarının eklenmesine imkân tanınmıştır. 7. ve 8. sorularda personelin mikroservis mimarisine hazır olup olmadığını öğrenmek amaçlanmıştır. Bu sorular da çoktan seçmeli sorulardır. 9. ve 10. sorular bu değişime karşı algı ve tutum değerlendirmek istenmiştir. Ayrıca üzerinde çalıştıkları yazılımların ve birlikte çalıştığı ekiplerin uygunluğunu değerlendirmeleri istenmiştir. 9. soru çoktan seçmeli ve açık uçlu bir sorudur. 10. soru Likert ölçeğine göre hazırlanmış çoktan seçmeli bir sorudur.

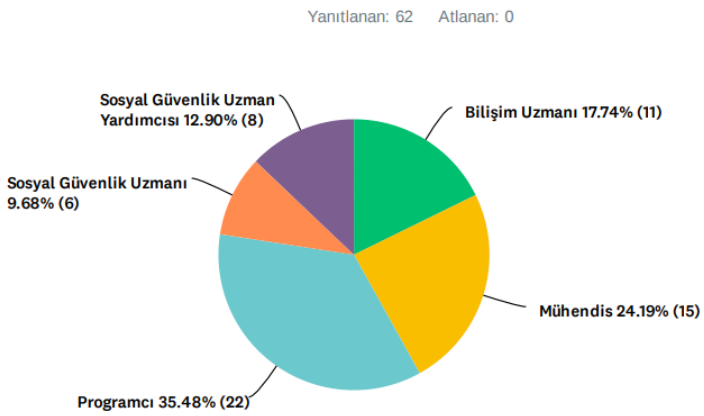
Anket yanıtlarının anonimliği üzerinden güvence verilerek e-posta üzerinden personele gönderilmiştir.

2.4. Sonuçlar ve Tartışma

Ankette sorulan her bir soru için bir grafik oluşturulmuş ve yorumlanmıştır. Her bir grafik ve soru aynı numaralandırılmıştır.

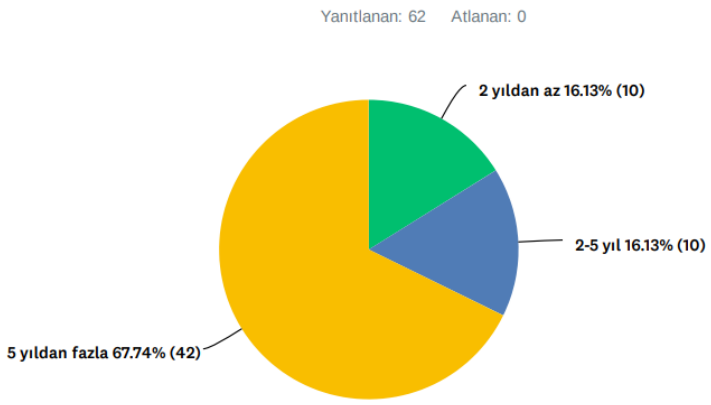
Ankete katılan 62 personelin unvanlara göre dağılımı *Grafik 1*'deki gibidir. Katılımcıların çoğunluğu programcılar (22), ardından mühendisler (15) ve bilişim uzmanlarından (11) oluşmaktadır.

Grafik 1: Katılan Personellerin Unvanlara Göre Dağılımı



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

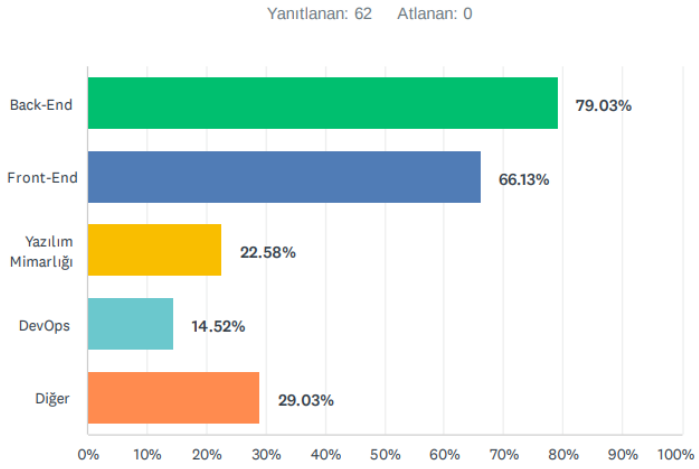
Grafik 2: Personelin Deneyim Yılı Dağılımı



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Ankete katılan personelin bilişim alanında %67,74 oranında yeterli derecede tecrübeli olduğu görülmektedir. Bu da anketten elde edilen sonuçların kuvvetli bulgular olacağına işaret etmektedir.

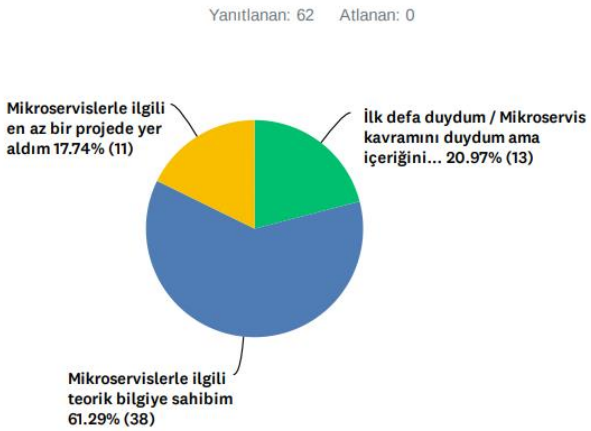
Grafik 3: Personelin Deneyimi Olan Roller



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Grafik 3 personelin deneyimlediği rollerin dağılımı verilmiştir. Yazılım alanı, birden fazla rolün aynı kişide olmasını gerektiren iç içe geçmiş bir alandır. Bu grafikte DevOps alanında personelin %14,52'ünün bilgisi olduğu görülmektedir. Mikroservis mimarisi ile uygulama geliştirilirken ve yönetirken DevOps teknolojilerini bilmek gerekmektedir. Bu alandaki bilgi eksikliği mikroservis mimarisinin uygulanmasını zorlaştırma potansiyeline sahiptir. Bu alanda gerekli eğitimler verilerek olası geçiş için önlemler alınmalıdır.

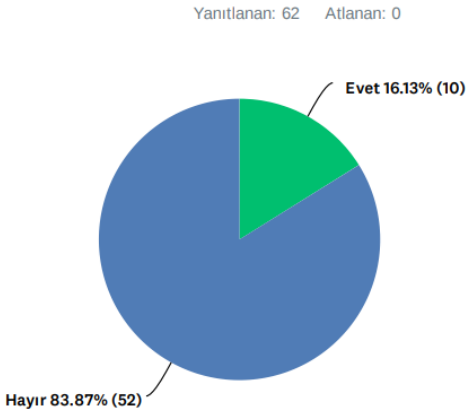
Grafik 4: Personelin Mikroservis Mimarisi Bilgisi



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Grafik 4'te personelin sadece %17,74'ü en az bir projede yer aldığını belirtmiştir. Tecrübe yılının bu kadar fazla olmasına rağmen mikroservis mimarisi ile geliştirilmiş bir projede çoğu kişinin yer almaması bu mimarinin yazılım piyasasında yeni olmasından kaynaklanmaktadır. Ayrıca yazılım dağıtım ve yönetiminde kullanılan araçların yeni yeni mükemmelleşmesi yönetilmesi zor olan mikroservis mimarisinin sonradan yaygınlaşmasını normal kılmaktadır.

Grafik 5: Personelin Mikroservis Mimarisi Proje Tecrübesi

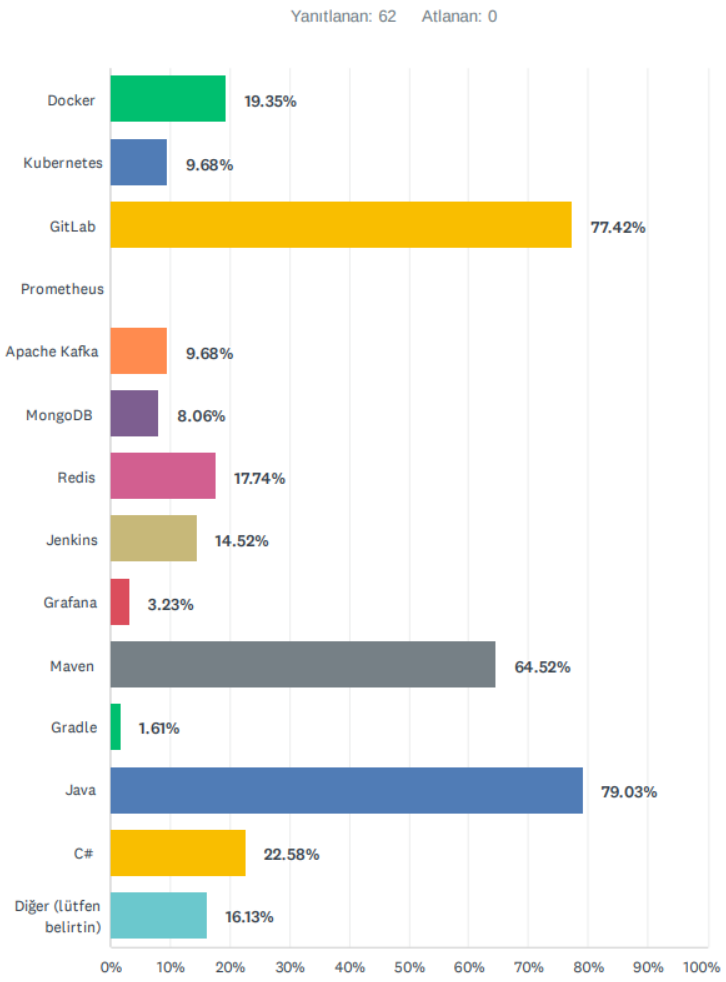


Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Grafik 5 ve *Grafik 4*'e ait sorular birbirini kapsayan ve onaylayan yapıdadır. İki grafik arasındaki 1 kişiden kaynaklanan tutarsızlık, *Grafik 4*'te **-Mikroservislerle ilgili en az bir projede yer aldım-** ve *Grafik 5*'te proje tecrübeniz var mı sorusuna **-Evet-**

olarak belirtmesi, personelin yorum farklılıklarından kaynaklanabilen göz ardı edilebilecek bir durum olarak değerlendirilmiştir.

Grafik 6: Personelin Kullandığı Teknolojiler

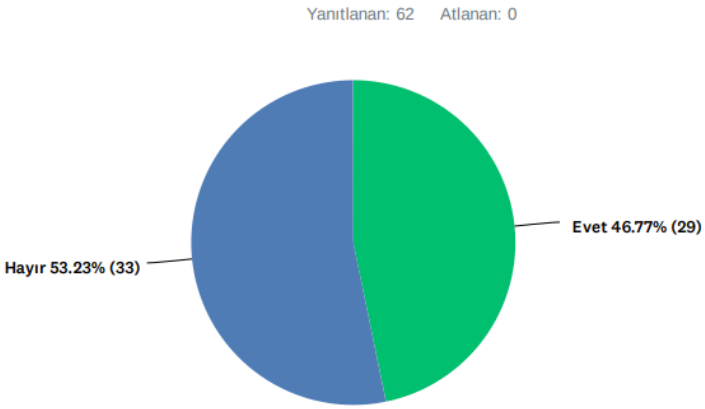


Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Personelin projelerinde kullandığı yazılım araçları *Grafik 6*'daki gibidir. Bu grafik kurumun mevcut teknoloji ortamına ilişkin içgörüler sunmaktadır. Personelin cevapları arasında, mikroservis mimarisinde uygulama oluşturmak ve yönetmek için sektör genelinde yaygın olarak benimsenen Docker, Kubernetes, GitLab, Jenkins, Java ve C# yer almaktadır. Özellikle Docker ve Kubernetes'in kullanımının yaygınlığı, kurumun mikroservisleri dağıtmak ve ölçeklendirmek için temel bileşenler olan konteynerleştirme ve orkestrasyon teknolojilerini benimsediğini göstermektedir. Java kullanım oranının %79,03 olması kurumda olası bir geçişte mikroservislere

adaptasyonunun kolay olacağını göstermektedir. Çünkü bu tezde de geliştirme yapılırken yararlanılan Spring Cloud gibi bir kütüphane avantajına sahiptir.

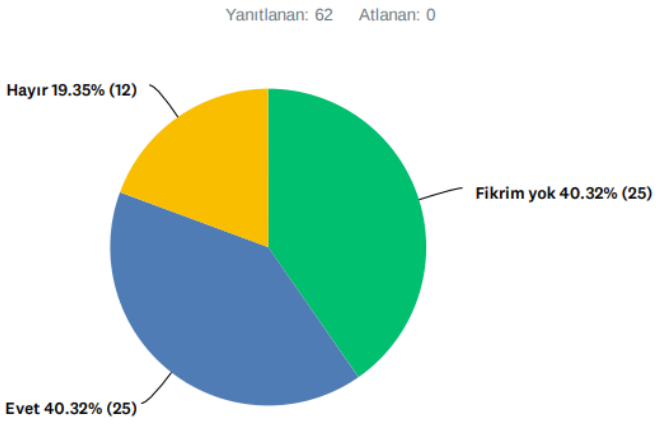
Grafik 7: Mikroservislerin Öğrenimi ve Uygulanması için Zaman ve Kaynak Kullanılabilirliği



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Grafik 7'de ankete katılanların %53,23'ü mikroservis mimarisini öğrenecek ve uygulayacak zamana sahip olmadığını belirtirken geriye kalan %46,77'lik kısım gerekli zamana sahip olduğunu belirtmiştir. Bu sonuç, iş yükü ve kaynak tahsisi gibi sorunlara işaret ediyor olabilir. Geçiş yapılacak birimde yeni bir mimariye geçişin sorunsuz bir şekilde uygulanabilmesi için yeterli uzman personel ve mevcut personel için gerekli eğitim sağlanmalıdır.

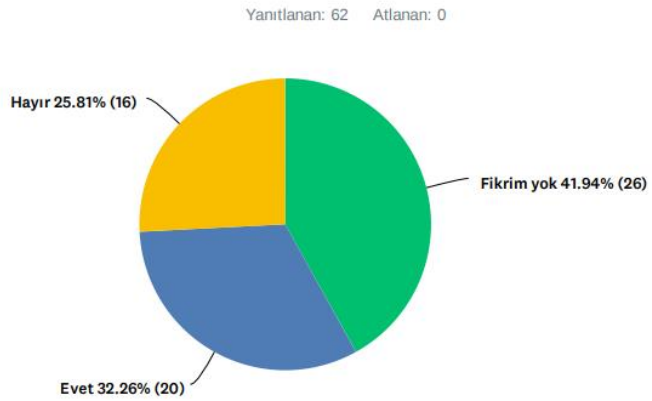
Grafik 8: Mevcut Uygulamalar İçin Mikroservis Mimarisinin Benimsenmesinin Faydalarına İlişkin Perspektifler



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

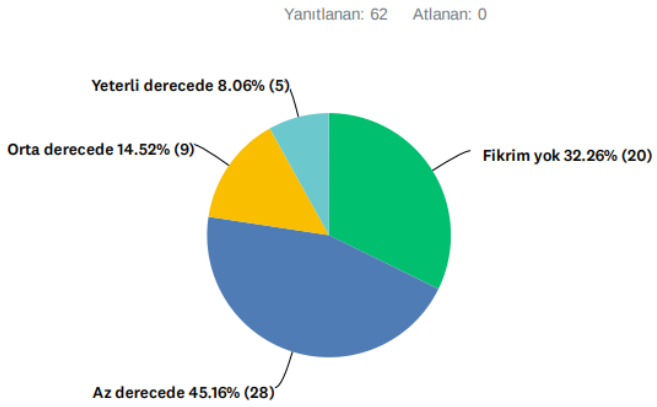
Sonuçlar, personellerin %40,32'sinin böyle bir geçişin faydalı olacağına inandığını, %19,35'inin katılmadığını ve %40,32'sinin emin olmadığını veya bir fikri olmadığını göstermektedir. Bu dağılım, katılımcıların benzersiz gereksinimlere ve zorluklara sahip farklı uygulamalar üzerinde çalışıyor olduğu göz önüne alındığında anlaşılabilir bir durumdur. Mikroservis mimarisi büyük, karmaşık ve sık güncelleme gerektiren uygulamalar için avantajlar sunmaktadır. Bunun aksine küçük ölçekli ve az bileşenli uygulamalar için uygulanması dezavantaj getirmektedir. %40,32 oranında geçişin avantaj getireceğini düşünen çoğunluk bir dönüşümün kaçınılmaz olduğuna işaret etmektedir. Kurum içinde dönüşüme ihtiyaç duyulan uygulamalar analiz edildikten ve gerekli altyapı sağlandıktan sonra pilot projeler seçilerek geçiş yapılmalıdır.

Grafik 9: Mikroservis Mimarisi İçin Altyapının Algılanan Yeterliliği



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Grafik 10: Mikroservis Mimarisi İçin Teknolojik Altyapı ve İnsan Kaynaklarının Algılanan Yeterlilik Derecesi



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Grafik 9 ve *Grafik 10* altyapı ve insan kaynakları bakımından kurumun mikroservis mimarisini benimsemeye hazır olup olmadığını değerlendirmektedir. *Grafik 9'*da katılımcıların %32,26'sı kurumun gerekli altyapıya sahip olduğuna inanırken, *Grafik 10'*da %22,58'i yeterli ve orta derecede hazır olduğunu belirtmiştir. %9,68'lik sapma *Grafik 10'*nun daha kapsayıcı bir soru olmasından yani insan kaynaklarını ele almasının bir sonucu olarak görülmektedir. Bu da yeterli teknolojik altyapının olduğu fakat personel bakımından sayı ve tecrübe eksikliğine işaret etmektedir. Mevcutta DevOps altyapısının sadece test ortamıyla sınırlı olması bu sonucu desteklemektedir.

Anket, SGK'nin mikroservis mimarisine yönelik mevcut hazırlık durumuna ilişkin genel bir bakış sunarak güçlü ve eksik alanların altını çizmektedir. BT personelinin önemli bir kısmı mikroservisler hakkında teorik bilgiye sahip olsa ve Docker ve Kubernetes gibi temel araçları kullansa da pratik deneyim olarak eksiklikler bulunmaktadır. Bu sonuç eğitim ve kaynak tahsisi konularında stratejik iyileştirmelere duyulan ihtiyacın altını çizmektedir.

3. ORGANİZASYONEL ETKİLER

3.1. Değişiklik Yönetimi

Mikroservis mimarisine geçiş, yalnızca teknik alanda değil, aynı zamanda kurumsal yapı ve kültürde de değişiklikleri içerir. Sorunsuz bir geçiş sağlayabilmek ve mikroservislerin tüm faydalarından yararlanabilmek için doğru bir yaklaşım önemlidir.

Mikroservisler yüksek derecede özerklik gerektirir. Her ekip kendi servislerinin tasarımı, geliştirilmesi ve dağıtımı ile ilgili kararlar alma yetkisine sahip olmalıdır. Monolitik geliştirme yaklaşımında analiz, tasarım, geliştirme (Front-End/Back-End) ve test ekipleri hepsi kendi alanıyla ilgili ekipte işlerini gerçekleştirirler. Mikroservis mimarisinde ise ekibin sorumlu olduğu uygulamanın bütün yazılım yaşam döngüsü aşamaları ekip tarafından yapılır. Mevcut durumda kurumda kullanılan yaklaşım yazılım ekibinin bütün aşamalarda var olduğudur. Bunun olası bir mikroservis mimarisine geçişi kolaylaştıran bir etken olduğu görülmektedir. Ayrıca bu yaklaşım iş ihtiyaçlarının ve hedeflerin uyumlu olmasını sağlar.

Mikroservisler dağıtık uygulamalardır. Çalışanlara uygulamaların yapısı nedeniyle mikroservis mimarilerinde daha yaygın olan ağ sorunlarını anlamaları için gerekli eğitimlerin verilmesi gerekmektedir. Bunun yanında CI/CD ve konteynerizasyon teknolojilerindeki becerileri geliştirmek için de eğitim programları uygulanmalıdır.

Mikroservisler genellikle iş yetenekleri etrafında düzenlendiğinden DDD'yi anlamak ve uygulamak çok önemlidir, bu da iş ihtiyaçlarına göre servis sınırlarının nasıl belirleneceği konusunda eğitim gerektirir. Ayrıca, DevOps ilkeleri, araçları ve uygulamaları konusunda tüm IT personelinin kapsamlı bir şekilde eğitilmesini içeren bir DevOps eğitimine de ağırlık vermek gerekmektedir.

Ekipler daha merkezi olmayan hale geldiği için etkili iletişim kanalları oluşturulmalıdır. Ekipler arasında iş birliğini ve bilgi paylaşımını kolaylaştıran araçlar kullanılmalıdır.

3.2. Paydaşlara Etkileri

Yazılım mimarisi değişiminin hem DevOps hem de geliştirici ekipleri üzerinde etkileri vardır. Mikroservis mimarisine geçiş, bu ekiplerin çalışma, iş birliği yapma ve yazılım teslim etme biçimlerini etkiler. Paydaşlar üzerinde etkileri anlamak, geçişi etkili bir şekilde yönetmek ve mikroservislerin potansiyel faydalarından yararlanmak için gereklidir.

Bir kurumda mikroservis mimarisine geçiş DevOps tarafında iş yükünün ve karmaşıklığının artması anlamına gelmektedir. Bunun nedeni bir dağıtım ile devreye alınabilecek bir uygulamanın parçalara ayrılarak birden fazla dağıtım şeklinde devreye alınmasıdır. DevOps ekibinin artan sorumluluğu karşılayacak kapasiteye ulaşması gerekmektedir. Bunu gerçekleştirmenin yolu ise DevOps ekibinin mikroservis mimarisi için geliştirilmiş yazılım araçlarını öğrenmesi ve uygulamasıdır.

Mikroservis mimarisi hem DevOps hem de geliştirici ekipleri için önemli zorluklar ve artan sorumluluklar da gerektirmektedir. Değişen dinamikleri ve gereksinimleri iyi anlamak büyük önem taşımaktadır. Geliştiriciler iş alanlarına daha ayrıntılı bir şekilde odaklanmalı ve servisler arası iletişimin sağlam ve güvenli olmasını sağlamalıdır. Sonuç olarak, mikroservis mimarisine başarılı bir şekilde

adapte olmak yalnızca teknik değil, aynı zamanda daha fazla iş birliği ve sürekli öğrenmeye yönelik kültürel bir değişime de bağlıdır.

BEŞİNCİ BÖLÜM
MİKROSERVİS UYGULAMA ÖRNEĞİ

Bu bölümde, monolitik mimari ile geliştirilmiş olan web tabanlı Emektar4B uygulamasının bir bileşeni, mikroservis mimarisi kullanılarak yeniden tasarlanmış ve geliştirilerek uygulamaya entegre edilmiştir. Emektar4B'nin mikroservis mimarisi ile geliştirilmeye elverişli büyüklükte olması, eski teknolojiler ile geliştirildiği için modernizasyona ihtiyaç duyması, teknik borcunun birikmiş olması nedeniyle örnek olarak ele alınması uygun görülmüştür.

Mikroservis mimarisine dönüştürülmek üzere belirlenen bileşenin ise monolitik uygulama ile bağımlılıklarının az olması nedeniyle, bağımsız bir şekilde geliştirilip dağıtılmaya uygun olduğu tespit edilmiştir. Bu seçimin geçiş sürecinde karşılaşılabilecek olası zorlukları azaltacağı öngörülmüştür.

Ayrıca Emektar4B uygulamasının mikroservis mimarisine geçişi için ilk aşamalarda neler yapılacağı gösterilmiştir. Geçiş adımlarına başlamadan önce mevcut kullanılan teknolojilerden bahsedilmiş ve bunların geçişe etkisi tartışılmıştır. Geçiş aşamasında ise ilk mikroservisin geliştirilmesinden kullanıma alınmasına kadar olan aşamalardan bahsedilmiştir.

1. UYGULAMANIN MEVCUT DURUMU

Emektar4B, nesne yönelimli bir programlama dili olan Java ile geliştirilmiş web tabanlı bir uygulamadır. Çerçeve olarak Struts-1 kullanılmaktadır. Struts-1 MVC mimarisini benimsemiş bir çerçevedir. Bu sayede kodlar ayrıştırılarak kavranması daha kolay ve karmaşıklığı daha az web uygulamaları oluşturulmaktadır. Fakat Struts-1 eski bir teknoloji olduğu için günümüz modern çerçevelerinin yeteneklerine sahip değildir. İnternette yeterli kaynağı yoktur.

Uygulamada web tabanlı kullanıcı arayüzü oluşturmak için JSP kullanılmıştır. Bu teknoloji, geliştiricilerin HTML, XML veya diğer belge türlerine dayalı dinamik web sayfaları oluşturmasında kullanılır ve bu da kullanıcıların web tarayıcılarında içeriklerin gösterilmesini sağlar. Struts-1 gibi JSP de eski ve kullanımı azalan bir teknolojidir.

Uygulamada veritabanı olarak DB2 kullanılmaktadır. DB2, IBM tarafından geliştirilen verileri depolamaya, yönetmeye ve çeşitli işlemler yapmaya yarayan ilişkisel veritabanı yönetim sistemi teknolojisidir. Veri bütünlüğünü ve güvenliğini sağlamaktadır. Emektar4B'de kullanılan veriler bu teknoloji ile saklanmaktadır.

Uygulama içinden veritabanı ile ilgili işlemleri yapmak için bir ORM aracı olan hibernate teknolojisi kullanılmaktadır. Uygulama tarafında nesne olarak ele alınan veriler veritabanı tarafında ilişkisel olarak tutulmaktadır. Hibernate iki taraf arasındaki eşleştirme görevini yapar. Uygulama tarafında SQL kodu yazmadan standart bir şekilde veritabanı işlemleri yapılmasını sağlamaktadır.

WebSphere Application Server, IBM tarafından geliştirilmiş bir uygulama sunucusudur. Emektar4B bu uygulama sunucusunda barındırılmaktadır. WebSphere Application Server güvenli ve ölçeklenebilir bir altyapı sağlamaktadır. Fakat WebSphere Application Server ücretli bir sunucudur. Ayrıca Emektar4B uygulamasının Java 8 sürümü bağımlılığına neden olmuştur. Java 8 eski bir sürüm olduğundan ve desteği biteceğinden bu sürüm ile devam edilmesi güvenlik açıklarına ve yeni özelliklerin gelmeyecek olmasına neden olacaktır.

Kullanılan bu teknolojiler içinde Struts-1, WebSphere Application Server ve JSP'nin eski bir teknoloji olması ve modern teknolojilerin gerektirdiği yeteneklere ve uyuma sahip olmamasının dezavantajı yapılacak olan bu geçiş ile giderilebilir. Bu açıdan bakıldığında mikroservis mimarisine geçiş avantaj getirecektir. DB2 ve hibernate teknolojilerinin ise yapılacak olan mimari geçişte avantajı veya dezavantajı görülmemiştir. Mikroservislere geçiş sonrası kurumun belirleyeceği politikalar bu teknolojilerin seçiminde belirleyici olacaktır.

Emektar4B uygulaması monolitik bir yapıdadır. Uygulamanın mevcut durumu düşünüldüğünde strangler yaklaşımının sorunsuz bir geçiş için daha uygun olduğu değerlendirilmiştir. Mikroservis mimarisine geçiş durumunda farklı teknolojilere aşamalı bir şekilde geçilebilmesinin büyük bir avantaj sağlayacağı düşünülmektedir.

2. GEÇİŞ SENARYOSU

2.1. Kullanılacak Teknolojilerin Belirlenmesi

Programlama dili olarak Java kullanılacaktır. Java programlama dilinin kullanımının mikroservislerin geliştirilmesinde önemli avantajları olduğu görülmüştür. Java, kurum içinde en bilinen ve yaygın olarak kullanılan programlama dili olması nedeniyle sonraki yapılacak geçişlerde bu doküman bir kaynak görevi görebilir. Geliştirme ekipleri arasında Java deneyiminin daha çok olması üretkenliği artırıp mikroservis mimarisine daha sorunsuz bir geçişi sağlayacaktır. Ayrıca zamandan tasarruf sağlanacağı ve hata potansiyelini en aza indireceği düşünülmektedir.

Java çerçevelerinin mikroservis mimarisi desteğine sahip olması, seçimde bir başka nedendir. Java platformları, özellikle mikroservislerin geliştirilmesini, dağıtımını ve bakımını kolaylaştırmak için tasarlanmış Spring Boot çerçevesine sahiptir.

Spring Boot yapılandırma kolaylığı ve tanıdığı imkanlar sayesinde hızlı bir şekilde mikroservis projelerinin başlatılmasına olanak tanır. Kapsamlı kütüphane ve topluluk desteği vardır. Bu, mikroservislerin verimli çalışması için kritik öneme sahip olan yapılandırma ve REST API'leri için entegre destek gibi temel özellikler sağlamaktadır. Bu yüzden Spring Boot ve REST API ile geliştirme yapılacaktır. Ayrıca uygulamayı derlemek, bağımlılıkları yönetmek ve paketlemek için Apache Maven otomasyon aracı kullanılacaktır.

REST API, mikroservislerin iletişimi için kapsamlı desteğe sahiptir. Bu seçimde öncelikle REST teknolojisinin kullanım kolaylığı sağlayan ve geliştirme sürecini hızlandıran basit ve anlaşılır yapısı etkili olmuştur. REST herkes tarafından anlaşılan ve uygulanması kolay olan standart HTTP yöntemlerinden yararlanır. Bu basitlik, mikroservisler arası iletişim karmaşıklığını azaltarak onu mikroservis mimarisi için ideal bir seçim haline getirmiştir. REST API, mevcut sistemler ve araçlarla sorunsuz entegrasyon sağlayarak geliştirici verimliliğini artırır ve entegrasyon sorunları potansiyelini azaltır.

Seçilen diğer bir teknoloji Spring çatısı altındaki Spring Cloud çerçevesidir. Spring Cloud, yapılandırma yönetimi, servis keşfi, yük dengeleme ve dayanıklılık için temel araçlar sağlayarak mikroservislerin geliştirilmesini ve yönetimini basitleştiren bir çerçevedir. Spring Boot ile sorunsuz bir şekilde entegre olur ve geliştiricilerin daha az karmaşıklıkla ölçeklenebilir ve sürdürülebilir dağıtılmış sistemler oluşturmasını sağlar.

Geliştirme yapmak için kullanılan teknolojiler yukarıda bahsedildiği gibi olacaktır. Mikroservislerin paketlenmesi ve dağıtımını yapmak için kullanılacak teknoloji Docker olacaktır. Dağıtımını yaptığımız mikroservislerin yönetimi ise Kubernetes de yapılacaktır. Kurumda bu teknolojiler test ortamında kullanılmaktadır. Bu teknolojilerin seçilmesinin bir nedeni kurumda kullanılması diğeri ise kendi alanında standart haline gelmiş öncü teknolojiler olmasından dolayıdır.

Tercih edilen teknolojiler birbiriyle iç içedir. Dolayısıyla uyum ve entegrasyon sorunlarını en aza indirmek ve yararlanacak daha çok kaynak bulmak için birini tercih etmek genel olarak diğerlerini tercih etmeyi gerektirmiştir. Yazılım teknolojileri alanında uyumu ve desteği gözeterek teknolojiyi seçmek doğru olacaktır. Bir diğer önemli nokta ise burada bahsedilen teknolojiler içinde başka isimde teknolojiler barındırmaktadır ve çok katmanlı bir yapıya sahiptirler. Her bir teknoloji isminden bahsetmek yerine kapsayıcı olan seçilip anlatılmıştır.

2.2. Geçiş Çalışmaları

Emektar4B çok fazla bileşeni bünyesinde barındırdığı için birçok mikroservise ayrılabilir durumdadır. Bu çalışmada belli bir bileşen seçilip, uygulamadan ayrı bir şekilde tasarlanıp bir mikroservis olarak çalıştırılacaktır. Geçiş senaryosu için seçilen bileşenin diğer bileşenlerle minimum bağımlılık sergilemesinin başlangıç için ideal bir seçim olacağı düşünülmüştür. Böylece daha sorunsuz ve daha az maliyetli bir geçiş süreci mümkün olur.

Emektar4B bileşenlerinin kapsamlı ve ayrıntılı bir analizi, TALEP bileşeninin diğer bileşenlere göre daha az bağımlılık gösterdiğini ortaya koymaktadır. *Tablo 1*'de görüldüğü üzere TALEP bileşeninin veritabanı katmanında sadece KISI bileşeniyle bağlantısı vardır. Uygulama tarafında da aynı durum söz konusudur. Bu özellik, onu

özerk bir mikroservis olarak ayırma ve yeniden geliştirme için en uygun aday haline getirmiştir. Ayrıca bu, entegrasyon süreçlerinin zorluğunu da en aza indirmektedir.

Tablo 1: Talep Bileşeninin Alanları ve Bağımlılıkları

ALAN ADI	ALAN TİPİ	AÇIKLAMA
ID	NÜMERİK	Talep bileşenine ait alan
KISI_ID	NÜMERİK	Talep bileşeninin Kisi ile bağlantısını sağlayan alan
TALEPTARIHI	TARİH	Talep bileşenine ait alan
VARIDETARIHI	TARİH	Talep bileşenine ait alan
VARIDESAYISI	NÜMERİK	Talep bileşenine ait alan
TALEPTIPI	NÜMERİK	Talep bileşenine ait alan

Kaynak: Bu çalışma kapsamında hazırlanmıştır.

2.2.1. Yazılım Geliştirme Aşaması

Yeni oluşturulacak talep-mikroservisi TALEP varlıklarıyla ilgili tüm işlemleri yönetecektir. Bunlar şu şekilde sıralanabilir:

- ✓ Bir kişi için tüm talepleri listeleme
- ✓ Mevcut talepleri güncelleme
- ✓ Yeni talep oluşturma
- ✓ Talep tarihlerindeki değişiklikleri yönetme

Bu REST API aracılığıyla gerçekleştirilecektir. Spring Boot kullanarak proje yapısı oluşturulmuştur. Oluşturulan bu yapıda TalepController isimli denetleyici sınıfı HTTP isteklerini işler ve TalepService katmanına yönlendirir. TalepService iş mantığını içerir ve CRUD işlemlerini gerçekleştirmek için TalepRepository katmanıyla etkileşime girer. TalepRepository Spring Data JPA kullanarak veri erişim yöntemleri sağlar. Veriler, DB2 veritabanı içinde saklanmakta ve TalepRepository aracılığıyla erişilmektedir. Talep sınıfı ise veritabanı tablosunun uygulama katmanındaki karşılığıdır. application.properties talep uygulamasına özel veritabanı ayarlarının da bulunduğu uygulamanın konfigürasyonunun gerçekleştirildiği yerdir.

Şekil 8: Talep Mikroservisi Proje Yapısı

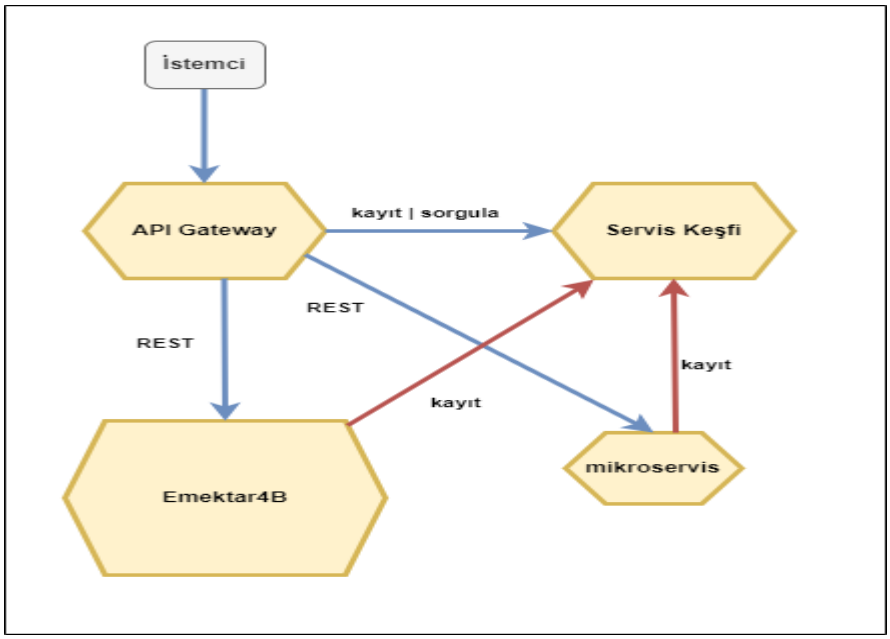
Kaynak: Bu çalışma kapsamında hazırlanmıştır.

talep-mikroservisi'nden sonra servis keşfi mekanizması gerçekleştirilmiştir. Servis keşfinin mekanizmasını gerçekleştirmek için Spring Cloud'un sunduğu Eureka Server kullanılmıştır. Eureka Server, bir ağdaki cihazların ve servislerin otomatik olarak tespit edilmesini sağlayan bir servis kayıt defteri görevi görür. naming-server adında Spring Boot projesi gerekli bağımlılıklar eklenerek oluşturulmuştur. application.properties dosyasındaki yapılandırmada, sunucunun adı ve bağlantı noktası belirtilmektedir. naming-server'ın yalnızca bir sunucu olarak çalışmasını sağlamak için istemci kaydı ve kayıt defteri getirme devre dışı bırakılır. Daha sonra mikroservisler Eureka sunucusuna kaydolacak şekilde yapılandırılır. Bu, application.properties dosyalarının Eureka sunucu URL'sini belirtecek şekilde değiştirilmesini ve bu URL adresinde bulunan kayıt defteriyle iletişim kurmasına izin verilmesini içerir.

Servis keşfi mekanizması bitince gerçekleştirilecek diğer adım API Gateway oluşturmaktır. API Gateway gelen istekleri karşılayacak ve gerekli uygulamaya iletecektir. Mevcut durumda talep-mikroservisi ve monolitik uygulama arasındaki istekleri yönlendirme işlevi yapacaktır. API Gateway kurmak için api-gateway adında Spring Boot projesi gerekli bağımlılıklar eklenerek oluşturulur. application.properties içinde Eureka sunucusuna kayıt ayarı yapılır. Böylece diğer uygulamalarla birbirinin adresini görmesi sağlanır. api-gateway içerisinde rota tanımları yapılır. Her rota ek filtreler içerebilen bir URI belirtir. Rotalar, Eureka'nın servis keşfi yeteneklerinden yararlanabilir. Son olarak api-gateway içerisinde loglama işlemi gerçekleştirilir.

Özetle sistem, monolitik Emektar4B uygulaması ve ondan ayırarak yeni bir mikroservis olarak geliştirilen ve kendine ait veritabanı olan talep-mikroservisi, bir API Gateway, bir Servis Keşfi olmak üzere 4 bileşenden meydana gelir. API Gateway merkezi giriş noktası olarak hareket eder. Kimlik doğrulama, yetkilendirme ve loglama işlemlerini yapar. Servis keşfi ise diğer uygulamaların adreslerinin tutulduğu bir kayıt defteri görevi görür. Dinamik olarak servis keşfi yapar. Eğer gerekli konfigürasyon yapılırsa yük dengeleme işlevi görebilir. Yani bir mikroservise aşırı istek gelmesi halinde mikroservisin örnekleri arasında isteklerin eşit şekilde dağıtılmasını sağlayabilir. talep-mikroservisi talep verilerinin manipülasyonundan sorumludur. Sistemin genel işleyişi bu şekildedir.

Şekil 9: Sistem İşleyişi



Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Sistemin geliştirilmesi ve işleyişi belirtildiği gibidir. Mevcutta çalışan monolitik sistemin bir bölümü sistemden ayrılıp ayrı bir uygulama gibi tasarlanmıştır. Bu uygulama yani talep-mikroservisi yine sisteme entegre edilip çalıştırılmıştır. Sonraki aşamalar ise geliştirilen mikroservis uygulamalarının paketlenip dağıtıma hazır biçime dönüştürülmesi aşamasıdır.

2.2.2. Yazılım Dağıtım Aşaması

Gerekli geliştirmeler yapıldıktan sonra diğer aşama uygulamaların dağıtıma hazırlanma aşamasıdır. Bu kısımda mikroservislerin dağıtık yapısını yönetebilmek için Docker ve Kubernetes gibi teknolojilerin devreye girmesi gerekecektir.

İlk kısım mikroservislerin(talep-mikroservisi, api-gateway, naming-server) Apache Maven ile derlenmesi ile başlar. Derleme sonucunda uygulamanın paketlenmiş bir çıktısı oluşmaktadır.

İkinci kısımda mikroservisler ile aynı dizinde bir yaml dosyası oluşturmak gerekir. Bu dosya birden fazla konteynerini tanımlamak ve yönetmek için oluşturulmuş bir konfigürasyon dosyasıdır. Veriler anahtar ve değer çiftleri olarak gösterilir. Docker-compose.yaml dosyası aşağıdaki gibidir.

Şekil 10: Docker-compose yaml dosyası

```yaml
version: '3.7'
services:
  talep:
    image: Mansur38/talep-microservice:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports: ["8000:8000"]
    networks: [emektar-network]
    depends_on: [naming-server]
    environment:
      EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://naming-server:8761/eureka
  api-gateway:
    image: Mansur38/api-gateway:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports: ["8765:8765"]
    networks: [emektar-network]
    depends_on: [naming-server]
    environment:
      EUREKA.CLIENT.SERVICEURL.DEFAULTZONE: http://naming-server:8761/eureka
  naming-server:
    image: Mansur38/naming-server:0.0.1-SNAPSHOT
    mem_limit: 700m
    ports: ["8761:8761"]
    networks: [emektar-network]
networks:
  emektar-network:
```

Kaynak: Bu çalışma kapsamında hazırlanmıştır.

Bu dosyadaki konfigürasyonda önemli noktalar şu şekilde özetlenebilir. services bölümünde uygulamayı oluşturan mikroservisler tanımlanmaktadır. Bu dosyada 3 hizmet vardır. Bunlar; talep-mikroservisi, api-gateway, naming-server'dır. image anahtarı barındırıldığı kayıt yerini göstermektedir. mem_limit, konteyner bellek sınırlarını; ports server ve konteyner tarafında kullanılacak portları; networks, mikroservisin kullanacağı ağı belirtmektedir. depends_on mikroservisin bağımlı olduğu diğer mikroservisleri gösterir. talep-mikroservisi ve api-gateway, naming-server'a bağımlıdır. Bağımlı olanlar, bağımlı olduğu mikroservisten sonra başlatılırlar.

Üçüncü kısım komut satırından gerekli komutların girilip projenin imajlarının oluşturulup sonrasında çalıştırılması işlemidir. Docker-compose.yaml dosyasının ve mikroservislerin bulunduğu dizine gelindikten sonra '`docker compose build`' komutu ile bu dosyada tanımlı mikroservislerin imajları oluşturulur. İmajları konteyner haline getirilip çalışması için '`docker compose up`' komutu kullanılıp uygulamalar çalıştırılabilir.

SONUÇ VE ÖNERİLER

Mikroservis mimarisi, günümüzde birçok büyük kurumun yazılım geliştirme süreçlerinde esneklik, ölçeklenebilirlik ve hızlı yanıt verme yeteneklerini artırmak için tercih ettiği bir mimari yaklaşım haline gelmiştir. Bu çalışma, mikroservis mimarisinin SGK'de doğru bir şekilde uygulanabilmesi için hem insan kaynaklarını hem de teknolojik altyapıyı göz önünde bulundurarak kurumun hazır olup olmadığını değerlendirmeyi ve monolitik bir mimariden mikroservis mimarisine geçiş için gereken adımların pratik bir taslağını sunmayı amaçlamıştır.

Bu kapsamda mikroservis mimarisine dair literatür araştırması yapılarak, mimarinin temel prensipleri, avantajları ve zorlukları hakkında bilgi toplanmıştır. Monolitik ve SOA mimarisi ile karşılaştırmalar yapılmış, bu mimariler arasındaki farklar incelenmiştir.

SGK'de kullanılan yazılım teknolojileri tespit edilerek bu teknolojiler arasında programlama dilleri ve entegrasyon araçları incelenmiştir. Bunların ne ölçüde mikroservis mimarisinin kullanılmasına katkıda bulunacağı göz önüne alınarak değerlendirmeler yapılmıştır.

Mikroservis mimarisi, SGK'nin yazılım hizmetlerinin erişilebilirliğine ve hızlı güncellenmesine büyük katkı sağlayabilecek bir potansiyele sahiptir. Bu mimari sayesinde, yazılım hizmetleri daha hızlı güncellenerek yazılım değişikliği gerektiren mevzuat değişikliklerine daha çabuk yanıt verebilir.

SGK'nin mevcut altyapısı, mikroservis mimarisine geçiş için gerekli donanım, işlemci gücü ve ağ kapasitesine sahiptir. Ancak bu geçişin başarılı bir şekilde uygulanabilmesi için, gerekli personel temin edilmeli ve/veya mevcut personele mikroservis mimarisi hakkında eğitim verilmelidir.

Mikroservis mimarisi ile geliştirilmiş bir uygulamanın verimli bir şekilde çalışabilmesi için DevOps yaklaşımı zorunludur. Mevcut durumda, DevOps

teknolojileri sadece test ortamında kullanılmaktadır. DevOps yaklaşımının tüm ortamlarda etkin bir şekilde kullanılması gerekmektedir, böylece süreçler arası entegrasyon ve otomasyon sağlanarak operasyonel verimlilik artırılabilir.

Monolitik bir uygulamanın bir bileşeninin mikroservis mimarisine dönüştürüldüğü bir vaka çalışması gerçekleştirilmiştir. Vaka çalışmasında, eski sistemin parçalarını kademeli olarak mikroservislerle değiştiren Strangler yaklaşımı benimsenmiştir. Eski teknolojiler ile geliştirilmiş EMEKTAR4B uygulaması incelenmiştir. Kurumsal yazılımların ömürlerinin yaklaşık 15 yıl olduğu ve bu süreyi doldurmak üzere olan EMEKTAR4B uygulamasının ihtiyaçları karşılamakta zorlandığı tespit edilmiştir. Büyük bir uygulama olduğu göz önünde bulundurulduğunda mikroservis mimarisi ile yeniden geliştirilmesinin uygun olduğu görülmüştür. Yapılan inceleme sonucunda EMEKTAR4B'de bulunan talep bileşeni ayrı bir mikroservis olarak yeniden yazılmıştır. Talep bileşeninin seçilmesinin nedeni az bağımlılık içerdiğinden dolayı hata ile karşılaşılma riskinin az olmasıdır. Öncelikle talep mikroservis'i Spring Boot kullanılarak geliştirilmiştir. Ardından mikroservislerin nerede bulunduğunu dinamik olarak bulmaya yarayan Servis keşfi mekanizması bağımsız bir yapı olarak oluşturulmuştur. Servis keşfi'nin geliştirilmesinden sonra api-gateway denilen dış dünyadan gelen isteklerin yönlendirilmesini sağlayan yapı oluşturulmuştur. Son olarak monolitik uygulama içerisinde dahili bir biçimde çalışan talep bileşeni devre dışı bırakılmış ve gerekli konfigürasyon ayarları yapılarak sistem çalıştırılmıştır. Böylece mikroservis de dahil olduğu hibrit yapı sağlanmış ve mikroservis mimarisine geçiş aşamasının önemli safhası gerçekleştirilmiştir.

Ayrıca vaka çalışmasında, SGK yazılım uygulamalarının eski teknolojilere olan bağımlılığının, yeni teknolojilerin entegrasyonunu kolaylaştıran bir model olan mikroservis mimarisi kullanılarak azaltılabileceği gözlemlenmiştir.

Yapılan anket çalışmasında SGK yazılım uygulamalarında Java kullanımının yaygın olduğunu göstermiştir. Java, Spring gibi mikroservis mimarisini destekleyen modern bir çerçeveye sahiptir. Bu da mikroservis mimarisine geçişte Java kullanımının bir avantaj olduğunu göstermektedir ve daha kolay bir geçişi mümkün kılacağını ortaya koymaktadır.

Bu çalışmada, örnek bir yazılım uygulaması seçilerek SGK'de mikroservis mimarisine geçiş süreci kapsamlı bir şekilde ele alınmış ve bu dönüşümün gerektirdiği adımlar detaylandırılmıştır. Mikroservis mimarisi, SGK için hem teknik hem de operasyonel açıdan büyük avantajlar sunma potansiyeline sahiptir. Ancak bu geçişin başarılı olabilmesi için, kurumun hem teknolojik altyapısını hem de insan kaynaklarını bu değişime uygun hale getirmesi gerekmektedir. Vaka çalışmasında ortaya konduğu üzere, kademeli bir geçiş süreci, SGK'nın mevcut sistemlerini modernize ederken operasyonel riskleri en aza indirme olanağı sağlamaktadır. Bu bağlamda, mikroservis mimarisine geçiş, SGK'nın yazılım hizmetlerini geleceğe taşımak için kritik bir adım olarak değerlendirilmiştir.

KAYNAKÇA

Adamson, C. (2023). Demystifying Kubernetes Kube-Proxy: Managing Behind the Scenes. Linkedin. https://www.linkedin.com/pulse/demystifying-kubernetes-kube-proxy-managing-behind-scenes-adamson-zgwlc adresinden alındı.

Armo. (2024). etcd in Kubernetes. Armo. https://www.armosec.io/glossary/etcd-kubernetes/ adresinden alındı.

Ayrancıoğlu, G. (2019). Monolitik Mimari ve Microservice Mimarisi Arasındaki Farklar. Medium. https://gokhana.medium.com/monolitik-mimari-ve-microservice-mimarisi-aras%C4%B1ndaki-farklar-bd89ac5b094a adresinden alındı.

Bass, L., Clements, P., & Kazman, R. (2012). Software Architecture in Practice (3. b.). Boston: Addison-Wesley.

Basumallick, C. (2022). What is Docker?. Spiceworks. https://www.spiceworks.com/tech/big-data/articles/what-is-docker/ adresinden alındı.

Cebeci, K., & Korçak, Ö. (2020). Design of an Enterprise Level Architecture Based on Microservices. Bilişim Teknolojileri Dergisi, 13(4), 357-371.

Chen, R., Li, S., & Li, Z. (2017). From Monolith to Microservices: A Dataflow-Driven Approach. Asia-Pacific Software Engineering Conference (APSEC) (s. 466-475). IEEE.

Cloud Academy Team. (2023). Docker vs Virtual Machines: Differences You Should Know. Cloud Academy. https://cloudacademy.com/blog/docker-vs-virtual-machines-differences-you-should-know/ adresinden alındı.

Çilingir, S. (2023). How to Convert Monolithic Architecture to Microservices Architecture: Strangler Pattern. Fimple. https://fimple.co.uk/how-to-convert-monolithic-architecture-to-microservices-architecture-strangler-pattern/ adresinden alındı.

docker.docs. (2024). Get Started with Docker. docker.docs. https://docs.docker.com/get-started/overview/ adresinden alındı.

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. IEEE Software, 33(3), 94-100.

Fowler, M. (2020). Domain-Driven Design. martinFowler.com. https://martinfowler.com/bliki/DomainDrivenDesign.html adresinden alındı.

Fowler, M., & Lewis, J. (2014). Microservices. martinFowler.com. https://martinfowler.com/articles/microservices.html adresinden alındı.

Growth Acceleration Partners. (2024). A Brief History of Microservices: Part I. Medium. https://medium.com/@wearegap/a-brief-history-of-microservices-part-i-958c41a1555e adresinden alındı.

Ilyukha, V. (2024). Monolith vs Microservices Architecture. Jelvix. https://jelvix.com/blog/monolith-vs-microservices-architecture adresinden alındı.

Jayasooriya, P. (2024). Monolithic vs SOA vs Microservices Architecture: A Java Perspective. Medium. https://medium.com/@lpramithamj/monolithic-vs-soa-vs-microservices-architecture-a-java-perspective-6d3d9fb26ac7 adresinden alındı.

Kızılpınar, D. (2021). Data Consistency in Microservices Architecture. Medium. https://dilfuruz.medium.com/data-consistency-in-microservices-architecture-5c67e0f65256 adresinden alındı.

kubernetes. (2015). Borg: Predecessor to Kubernetes. kubernetes. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/ adresinden alındı.

Lotfy, R. (2023). Domain-Driven Design (DDD). Linkedin. https://www.linkedin.com/pulse/domain-driven-design-ddd-ramadan-lotfy adresinden alındı.

Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing. In R. Sturm, C. Pollard & J. Craig (Eds.), Application Performance Management (APM) in the Digital Enterprise (s. 267-269).

Newman, S. (2015). Building Microservices. O'Reilly Media, Inc.

Pachghare, V. K. (2016). Microservices Architecture For Cloud Computing. Mat Journals, 2(1), 1-13.

Pantic, N. (2022). Kubernetes Architecture Diagram. ClickIT. https://www.clickittech.com/devops/kubernetes-architecture-diagram/ adresinden alındı.

Perry, D. E., & Wolf, A. L. (1992). Foundations for the Study of Software Architecture. Software Engineering Notes, 17(4), 41.

Phelan, T. (2021). Why CEOs Should Learn the Kubernetes Way of Thinking. CIO. https://www.cio.com/article/189372/why-ceos-should-learn-the-kubernetes-way-of-thinking.html adresinden alındı.

Red Hat. (2022). IaaS vs PaaS vs SaaS. Red Hat. https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas adresinden alındı.

Sanchez, J. (2024). A Brief History of Microservices: Part I. Medium. https://medium.com/@wearegap/a-brief-history-of-microservices-part-i-958c41a1555e adresinden alındı.

Sommerville, I. (2010). Software Engineering (9. b.). Pearson Education Limited.

Sosyal Güvenlik Kurumu. (2024). Bilişim Yönetim Sistemi. https://uygyonetim.sgk.intra/UygYonetimV2/faces/Pages/DisPaydaslarlaVeriPaylasanUygulamalar.xhtml adresinden alındı.

Yasar, K. (2023). Cloud Computing. TechTarget. https://www.techtarget.com/searchcloudcomputing/definition/cloud-computing adresinden alındı.