T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

KOD KALIP ANALİZİ YÖNTEMLERİ İLE MİKROSERVİS MİMARİLERİNDE İYİLEŞTİRME ÖNERİLERİ

TUĞRUL AŞIK

YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ PROGRAMI

DANIŞMAN
YRD. DOÇ. DR. YUNUS EMRE SELÇUK

İSTANBUL, 2017

T.C.

YILDIZ TEKNİK ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

KOD KALIP ANALİZİ YÖNTEMLERİ İLE MİKROSERVİS MİMARİLERİNDE İYİLEŞTİRME ÖNERİLERİ

Tuğrul AŞIK tarafından hazırlanan tez çalışması 09.11.2017 tarihinde aşağıdaki jüri tarafından Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilimdalı'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Tez Danışmanı

Yrd. Doç. Dr. Yunus Emre SELÇUK

Yıldız Teknik Üniversitesi

Jüri Üyeleri

Yrd. Doç. Dr. Yunus Emre SELÇUK

Yıldız Teknik Üniversitesi _____

Prof. Dr. Oya KALIPSIZ

Yıldız Teknik Üniversitesi _____

Prof. Dr. Nadia ERDOĞAN

İstanbul Teknik Üniversitesi _____

ÖNSÖZ

Bu tez, Mikroservis Mimari olarak tasarlanan ve geliştirilen uygulamalarda, yazılım kalitesinin yükseltilmesi, mimari yaklaşıma uygunluğunun tespit edilmesi ve kötü kod geliştirme pratiklerden kaçınılması için yeni metrik tanımları ve yaklaşımlar içermektedir. Tanımlanan metrik ve öne sürülen yaklaşımların, statik kod seviyesinde otomasyonel analizi için MISKAA adında analiz aracı geliştirilmiş ve analiz aracından elde edilen raporlar sonucunda çıktıların uygulamayı iyileştirme doğrultusunda fayda sağlayıcı olması amaçlanmıştır.

Öncelikle tez konusunu seçerken isteklerimi göz önünde bulundurup bana yardımcı olan tez danışmanım Yrd. Doç Dr. Yunus Emre SELÇUK'a teşekkürlerimi sunarım. Yüksek lisans çalışmalarına teşvik eden Merkezi Kayıt İstanbul A.Ş ve şirket yöneticileri ile değerli iş arkadaşlarıma, tez sürecindeki desteklerinden dolayı değerli arkadaşlarım Uğur SEZER, Akif ONUR, Rüstem Can AYGÜN ve İrfan BULUT'a ayrıca teşekkürler. Tüm eğitim hayatım boyunca her türlü desteği hiçbir zaman esirgemeyen sevgili aileme en içten teşekkürlerimi sunar ve nicelerini bir borç bilirim.

Bu tez, yüksek lisans çalışmalarım devam ederken ebediyete uğurladığımız sevgili dedem Ahmet ÇELİK'e ithaf edilmiştir.

Kasım, 2017

Tuğrul AŞIK

İÇİNDEKİLER

Sayfa

SİMGE LİSTESİ

CC	İstemcilerin sayısı
RC	Kaynakların sayısı
S	Toplam servis büyüklüğü
UCC	Kullanılmayan İstemci Sayısı
URC	Kullanılmayan Kaynak Sayısı

KISALTMA LİSTESİ

API	Application Programming Interface
AST	Abstract Syntax Tree
CC	Client Count
HTTP	Hyper Text Transfer Protocol
IP	Internet Protocol
LURI	Long URI
MISKAA	Mikroservis Statik Kod Analiz Aracı
RC	Resource Count
S	Size
SOA	Service Oriented Architecture
SURI	Static URI
UCC	Unreachable Client Count
URC	Unused Resource Count
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
QoS	Quality of Service

ŞEKİL LİSTESİ

Sayfa

ÇİZELGE LİSTESİ

Sayfa

ÖZET

KOD KALIP ANALİZİ YÖNTEMLERİ İLE MİKROSERVİS MİMARİLERİNDE İYİLEŞTİRME ÖNERİLERİ

Tuğrul AŞIK

Bilgisayar Mühendisliği Anabilim Dalı

Yüksek Lisans Tezi

Tez Danışmanı: Yrd. Doç. Dr. Yunus Emre SELÇUK

Mikroservis mimari son dönemlerde yeni bir fenomen haline gelmeye başlamış olan mimari bir tasarımdır. Mikroservis mimarisi, dil bağımsız ve dinamik olarak ölçeklenebilir bir altyapı sunmaktadır. Bu mimari, temelde her bileşenin kendi içinde bir bütün olduğu ve bu bileşenlerin dağıtık olarak konumlandığı bir desen olarak tanımlanabilir. Yazılım uygulamalarının bağımsız olarak konuşlandırılabilir servis grupları halinde nasıl tasarlanmasıyla ilgili yöntemler sunmaktadır. İş yapma kabiliyeti, otomasyonel dağıtım, otomasyonel süreçler, akıllı erişim noktaları ve bileşene özgü ve merkezi olmayan veri yönetimi mikroservis mimarinin benzer karakteristik özelliklerdendir. Diğer yandan, bu mimari tarzın kesin ve tamamıyle olgunlaşmış bir tanımı bulunmamaktadır. Mimarinin bahsedilen karakteristik avantajları, ağ üzerinden servislerin keşfedilmesi, güvenlik yönetimi, iletişim optimizasyonu, veri paylaşımı ve performans gibi zorluklarları beraberinde getirmektedir. Ayrıca, ekip büyüklüğü, şirket kültürü, kişisel beceriler, üretkenlik vb. etmenler de yazılım geliştirme sürecinde ve yazılım ürününde bir etkiye sahiptir. Çalışma kapsamında, mikroservis mimari için tanımlı karakteristik özellikler ve farklılıklara göre, yazılım kalitesini ve mimariye uygunluğunu ölçmek için yeni metrik ve yaklaşımlar tanımlanmıştır. Bu metrik ve yaklaşımlar servis boyutunu ölçmek, servisler arası iletişimi analiz etmek ve yazılımı iyileştirmek için yapılmaması gereken pratikleri keşfetmek olarak kategorize edilebilir. Mikroservis mimari tabanlı geliştirilen bir yazılım ürünü üzerinde, tanımlanan metriklere ve yaklaşımlara uygunluğu kontrol etmek için, MISKAA statik analiz aracı geliştirilmiştir. Tanımlı metrikleri ve yaklaşımları değerlendirmek için örnek bir bilet

satış uygulaması uyarlanmıştır. Analiz sonuçlarının yazılımı iyileştirme ve hataların erken teşhisi konusunda yol gösterici olacağı düşünülmektedir.

Anahtar Kelimeler: Mikroservis mimari, yazılım metrikleri, yazılım kural seti, yazılım kalitesi, statik analiz aracı

ABSTRACT

IMPROVEMENT SUGGESTIONS FOR SOFTWARE BASED ON MICROSERVICE ARCHITECTURE WITH CODE ANALYSIS TECHNIQUES

Tuğrul AŞIK

Computer Engineering Department

MSc. Thesis

Adviser: Asist. Prof. Yunus Emre SELÇUK

Microservice is an architectural style that has recently started gaining popularity to become a new architectural phenomenon. Microservice architecture provides new opportunities to deploy scalable, language free and dynamically adjustable applications. Microservice based architecture is defined as a software architecture pattern for development of distributed applications, where the application is comprised of a number of smaller independent components; these components are small application in-themselves. This architecture style has gained popularity over the last few years to describe a particular way of designing software applications as suites of independently deployable services. There are certain common characteristics around organization around business capability, automated deployment, built released with automated processes, intelligence in the endpoints, and decentralized control of languages and data. On the other hand, there is no precise definition of this architectural style and mentioned benefits come with challenges, such as discovering services over network, security management, communication optimization, data sharing and performance. Furthermore, team size, culture, skills, productivity etc. have an impact on software development. Unfortunately, there is no strict rule about these challenges; there are some practices that have been proven in real-world systems. According to this characteristic features and tradeoffs, defining new metrics and practices to measure software quality and suitability for microservice practices were studied. New metrics were defined to measure service size and inter-service communication. In order to check compliance with these metrics, static analysis tool

which name is MISKAA to analyze software based on microservice architecture were developed. A sample ticket selling application to evaluate these metrics and practices were implemented. Analysis results would be helpful for project team to improve software quality and to detect bugs or bad practices.

Keywords: Microservice architecture, software metrics, software policies, software quality, static analysis tool

BÖLÜM 1

GİRİŞ

1.1 Literatür Özeti

Martin Fowler ve James Lewis tarafından gündeme getirilen mikroservis mimari stili büyük çaplı uygulamalara farklı bir mimari bakış açısı kazandırmıştır. Amazon, Netflix, Spotify, SoundCloud gibi günde milyonalarca kişi tarafından kullanılan alt yapı sunan firmalar uygulamalarında ve verdikleri hizmette bu yapıya kendilerini uyarlamaktadırlar. Türkiye'de de Merkezi Kayıt İstanul A.Ş, ETS Tur, n11.com gibi yazılım alt yapılarını sürekli güncel tutan büyük firmalar, mikroservis mimarisine uygun ürünler geliştirmeye başlamışlardır. Dönüşüm işlemi hem kültürel hem teknik hem de altyapı ile ilgili değişiklikleri ve zorlukları beraberinde getirmektedir. Bu derece geniş çaplı bir değişimin içinde yazılım kalitesini yönetmek ve geliştirilen ürünün belirtilen mikroservis mimari kıstaslarına uygunluğunu kontrol etmek gerekmektedir. Mikroservisler arası iletişim, keşfedilebilirlik ve işlem loglarının işlenebilmesi için bazı yaklaşımlar ortaya atılmıştır ve bu amaçla Consul, Eureka gibi ürünler geliştirilmiştir. Kalite analiz yöntemleri ve yazılım sürekliliğini sağlamak adına AWS Cloud Watch, Promethous ve Sonar Qube gibi araçlar tanımlı metrikler çerçevesinde analiz ve monitörleme yapabilmektedir. Kendi içinde tek bütün haline çalışan uygulamalarda birçok kalite analiz metriği ve bu metrikleri analiz edecek araç bulunmasına rağmen, dağıtık olarak bu işlemleri yerine getirecek bir metrik grubu ve bu metrik grubunu işleyebilecek bir analiz aracı bulunmamaktadır. Mikroservis uygulamaları için dağıtık olarak tasarlanmış servislerde, kalitenin analiz edilebileceği, servisler arası iletişimin statik kod seviyesinde incelenebileceği yeni metrik ve yaklaşımlara ihtiyaç olduğu görülmüş ve tez çalışması bunun üzerine inşaa edilmiştir.

1.2 Tezin Amacı

Mikroservis mimari tabanlı geliştirilen bir uygulamanın, mimariye uygunluğunu kontrol etmek, tanımlı ve mimarinin tasarımıyla ilgili henüz tam olarak netleştirilmemiş kriterlerle ilgili metrikler öne sürmek ve geliştirme sırasında yapılabilecek anti-pattern davranışları tespit etmektir. MISKAA statik kod analiz aracı tanımlı metriklerin analizi için geliştirilecek ve çıkan sonuçlar uygulama kalitesini yükseltmek amacıyla kullanılacaktır.

1.3 Hipotez

Mikroservis mimari tabanlı uygulamalarda, servisler arası ilişkiler, servis büyüklükleri ve geliştirme sırasında yapılabilecek anti-pattern davranışlar için metrikler ve pratikler tanımlanabilir. Belirlenen bu değerler statik kod analiz yöntemleri ile mikroservis tabanlı uygulamalar üzerinde analiz edilebilir. Analiz sonrası oluşturulan raporlar ile yazılım kalitesi iyileştirilebilir.

BÖLÜM 2

MONOLİTİK, SOA VE MİKROSERVİS MİMARİ

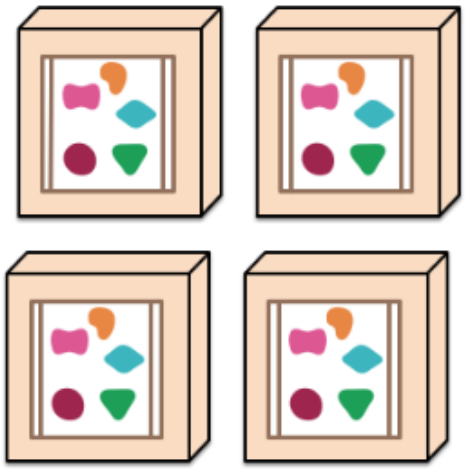
Yazılım uygulama mimarisi, performans, güvenlik ve yönetilebilirlik gibi ortak nitelikleri optimize ederken, tüm teknik ve operasyonel gereksinimleri karşılayan yapılandırılmış bir çözümü tanımlama işlemidir. Çok çeşitli faktörlere dayanan bir dizi karar içerir ve bu kararların her biri, uygulamanın kalitesi, performansı, sürdürülebilirliği ve genel başarısı üzerinde önemli etkiye sahip olabilir [1].

Philippe Kruchten, Grady Booch, Kurt Bittner ve Rich Reitman, Mary Shaw ve David Garlan'ın tanımını baz alarak 1996 yılında mimariyi şu şekilde tanımladılar; "Yazılım mimarisi, bir yazılım sisteminin organizasyonu hakkında, sistemin oluşturduğu yapısal öğelerin ve arabirimlerin seçimi de dahil olmak üzere önemli kararları kapsar. Bu unsurlar arasında sistemler arası tanımlı işbirlikleri, yapısal davranışlar ve daha büyük sistemler ile oluşturulan kompozisyonlar bulunmaktadır. Yazılım mimarisi aynı zamanda işlevsellik, kullanılabilirlik, esneklik, performans, anlaşılabilirlik, ekonomik ve teknoloji kısıtlamaları ve estetik kaygıları da içermektedir [1].

Uygulama mimarisi, kullanım durumlarını anlamak ve yazılımdaki bu kullanım durumlarını uygulama yollarını bularak iş gereksinimleri ve teknik gereksinimler arasında bir köprü kurmayı amaçlamaktadır. Mimarinin amacı, uygulamanın yapısını etkileyen gereksinimleri belirlemektir. İyi mimari, teknik çözüm ile iş tarafında oluşabilecek riskleri azaltmaktadır. İyi bir tasarım, donanım ve yazılım teknolojisinde zaman içinde oluşacak doğal kaymanın yanı sıra kullanıcı senaryoları ve gereksinimleri karşılayabilecek kadar esnek olarak nitelendirilebilir [1], [2], [3].

2.1 Monolitik Mimari

Monolitik mimariler, mimarinin ihtiyaç duyduğu tüm işlevleri bir araya getiren tek bir uygulama seviyesinde çalışan mimarilerdir. Mimari seviyede, bu, mimarinin en basit biçimidir. Çünkü diğer mimari stiller kadar aktör içermez [2], [3].



Şekil 2. 1 Monolitik mimari

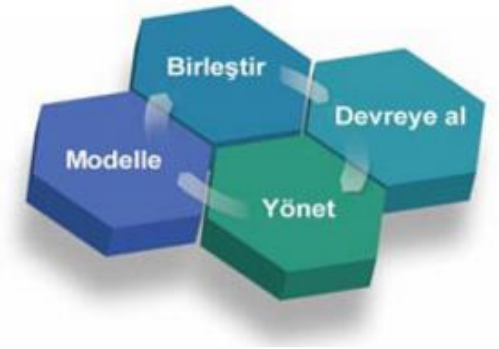
Mimarinin avantaj ve dezavantajlarını keşfetme noktasında karakteristikleri vardır. Bağımlılıklar ve büyüdükçe hantal bir yapıya bürünme, monolitik mimarinin dikkat çeken karakteristiklerdendir [2], [3].

Karakteristik Özellikleri

- Hızlı geliştirilebilme ve ilk adaptasyonun kolay olması
- İyileştirilebilir entegrasyon
- Kullanıcı arayüzlerinde benzer kullanımlar
- Merkezi olması ve sorumluklarının tek bir noktada birleşmesi
- Sıkı bağımlılıklar
- Bakım işlemlerinin büyüdükçe zor hale gelmesi
- Kurulum maliyetlerinin yüksek olması
- Kod sahiplik derecesinin düşük olması
- Tümleşik Dağıtımın (Continuous Delivery) zor olması
- Versiyon yönetiminin uygulama büyüdükçe zorlaşması
- Değişimlerin tüm yapıyı etkilemesi

2.2 Servis Yönelimli Mimari

Servis yönelimli mimari (SOA – Servis Oriented Architecture), servis olarak adlandırılan, gevşek bağlı (Loosely Coupled), iri taneli (Coarse Grained) ve özerk (Autonoums) yapıdaki bileşenlere dayalı dağıtık sistemlerin geliştirilmesi için kullanılan mimari bir stildir. Geliştirilebilir, çevik, değiştirilebilir bileşenli, keşfedilebilir ve yeniden kullanılabilir servislerin web servis olarak uyarlanışıdır [4], [5].



Şekil 2. 2 SOA yönetim modeli

Karakteristik Özellikleri

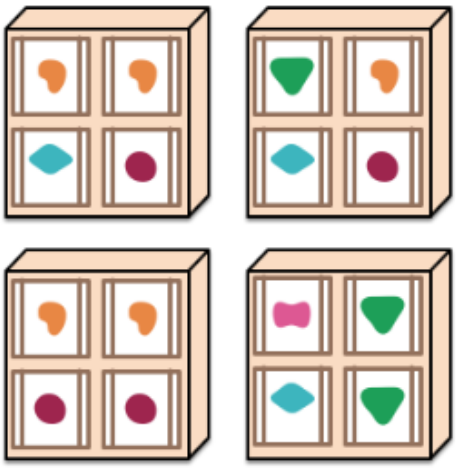
- Açık standartlı
- Mimari anlamda üretime uyumlu
- QoS'yi destekler
- Bağımsız çalıştırılabilir
- Keşfedilebilir
- Genişletilebilir
- Servis tabanlı soyutlanabilir
- Organizasyonel çevikliğe sahip

Yukarıda belirtilen özellikleri monolitik mimari karakteristikleri ile karşılaştırdığımızda birçok anlamda SOA mimarisinin durumu avantajlı hale getirdiğini söyleyebiliriz. Her ne kadar geliştirilmeye çalışılan yazılım ürünü ve oluşturulan çözüme göre sonuçlar değişse de büyük organizsyonlarda SOA daha fazla kabul gördüğü gözlemlenmektedir.

2.3 Mikroservis Mimari

Mikroservis mimari SOA mimarisinin bir parçası olmakla birlikte kendi içinde karateristiklere sahiptir. Martin Fowler ve James Lewis tarafından kısaca şu şekilde tanımlanmıştır; “Mikroservis mimari tarzı, her biri kendi içinde çalışan küçük paket parçaçıklardan oluşan bir uygulama bütününün geliştirilmesi için bir yaklaşımdır. Hafif olarak nitelendirilebilecek yöntemlerle, servisler birbirleriyle bir API ile iletişim kurarlar ve bu API genellikle HTTP üzerine kuruludur. Bu servisler, belirli bir iş güdümü etrafında oluşturulmuş olup, tamamen otomatik dağıtım mekanizmalarıyla bağımsız olarak konuşlandırılabilirler. Farklı programlama dillerinde yazılmış olabilirler ve farklı veri depolama teknolojilerini kullanabilirler ve servislerin merkezi olarak yönetilmesine yönelik bir kapsam vardır [6].

Belirli bir dağınık yazılım mimarisi tarzını tanımlayan mikro servisler, eksiksiz bir sistem oluşturmak için birlikte çalışan küçük, bağımsız olarak konumlandırılabilen birimlerdir. Mikroservisler web'de ve bulutta yaşayabilir ve her türlü veri ile çalışabilir.



Şekil 2. 3 Mikroservis mimari

Karakteristik Özellikleri

- Servisler halinde bir bütün oluşturma
- İş yetenekleri etrafında organize olma
- Projeden çok ürüne odaklanma
- Akıllı uç noktalar ve kukla bağlantılar

- Merkezi olmayan veri yönetimi
- Altyapısal otomasyon
- Altyapısal hatalara toleranslı tasarım
- Evrimsel tasarım

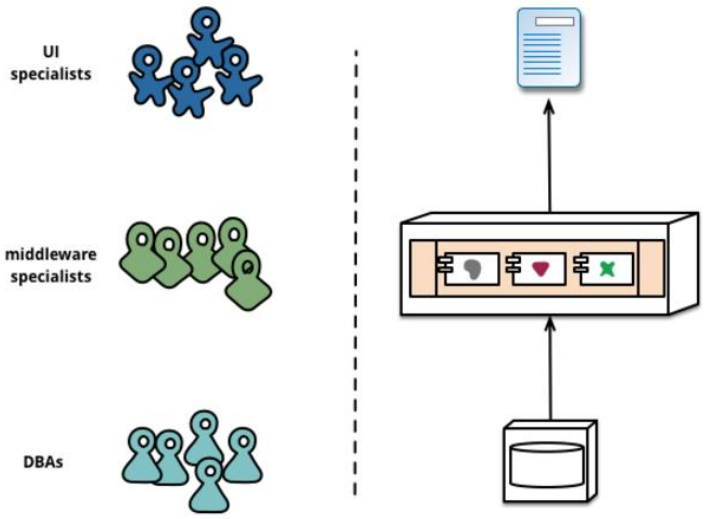
2.3.1 Servisler Halinde Bir Bütün Oluşturma

Uzun yıllardır yazılım endüstrisi lego parçaları gibi takıp çıkarabileceği ve birbirinden bağımsız çalışabilecek bileşenler ile büyük organize yapılar kurma üzerine yaklaşımlar ortaya atmaktadır. Birçok kütüphane barındıran bir kod parçasına sahip bir yapıda yapılan küçük değişiklikler yapının ilgisiz farklı noktalarında da değişimlere sebep olabilmektedir. Monolitik mimarileri bu kapsamda düşünüldüğünde herhangi bir yazılım geliştirmesi ihtiyacında bu geliştirme birkaç satır dahi olsa yazılımın tamamının güncellenmesi(derlenme ve sunulma) gerekmektedir. Mikroservis mimari bu tarz durumların kesin bir çözümü olmamakla birlikte bu durumları en aza indirmeyi amaçlamaktadır. Birbiriyle HTTP API'leri üzerinden konuşan ve kendi içinde bir bütün olan küçük bileşenler ile az bağımlı bir organizasyon kurulması hedeflenmektedir [6, 8, 9]. Şekil 2.3'te her bir küreyi (servisi) kendi içinde bir bütün ve diğer kürelerle API'ler üzerinden konuşan bir organizyon parçası olarak hayal edilebilir.

2.3.2 İş Yetenekleri Etrafında Organize Olma

Bir uygulama daha küçük parçalara bölünme durumunda kırılımlara odaklanılır ve yazılım, arayüz geliştirme, veritabanı gibi ekipler kurgulanır ve çalışmalar yapılır. Şekil 2.4'te Conway'in Kanunu'nu temsil eden yapı özetlenmiştir.

Mikroservisler bu anlamda alamda farklıdırlar ve iş kapasiteleri etrafında konuşlanmış alt yapılara bölünürler. Alan bazında gruplanmış takımlardan çok çapraz işlevsel takımlara odaklanılır. Takım üyelerinde veritabanı uzmanlığı, yazılımcı geliştirme, proje yönetimi gibi işlevsellikler bulunur ve servisler bu organizasyon ile geliştirilir [6]. Şekil 2.5'te çapraz işlevsel takımlar ve yetkinlikler etrafında organize olmuş yapıyı bulabilirsiniz.

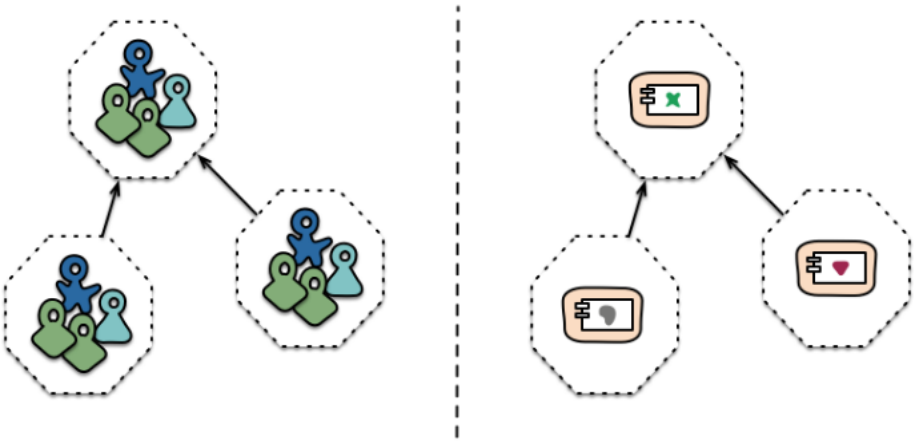


Şekil 2. 4 Conway'in iş bölümü organizasyonu

2.3.3 Projeden Çok Ürüne Odaklanma

Birçok geliştirme organizasyonunda projeye başlanır, proje teslim edilir ve daha sonrasında projenin bakımından sorumlu birimlere devredilir. Kısacası projeye başlayan takım işi bitince oyunda çekilir.

Mikroservis yaklaşımda takımın ürünü sahiplenmesi adına bundan kaçınılır. Amazon'un kısa bir tümceyle belirttiği gibi, "Sen inşaa ettin, sen çalıştır.", ürünün tüm sorumluluğu ve sahipliği geliştiren ve kurgulayan ekiplerdedir. Bu, üründe uzmanlaşmayı sağlar, müşteriyi anlamayı ve ürüne yeni özellikler kazandırmayı destekler [6].



Şekil 2. 5 Çapraz işlevsel takımlar ve servis kapsülleri

2.3.4 Akıllı Uç Noktalar ve Kukla Bağlantılar

Farklı süreçler arasında iletişim alt yapıları oluşturulurken, alt yapı görevi gören kısımların birçok mantıksal akışı yönettiği, yönlendirmerde bulunduğu, koreografiden sorumlu olduğu sistemler bulunmaktadır ve ESB bunun örneklerindendir [7].

Mikroservis yaklaşımda bu, yukarıda belirtildiği şekilde bir sistem oluşturulması yerine her türlü mantıksal ve işlevselliklerden uç nokta bileşenlerinde tasarlanmasıdır. Bu bileşenlerin sadece iletişiminden kukla bağlantılar sorumludur. Kısacası bir bileşen kendi kendine yetebilir ve sadece iletişim kurmak için basit bir bağlantı hattına ihtiyaç duyar [6], [8].

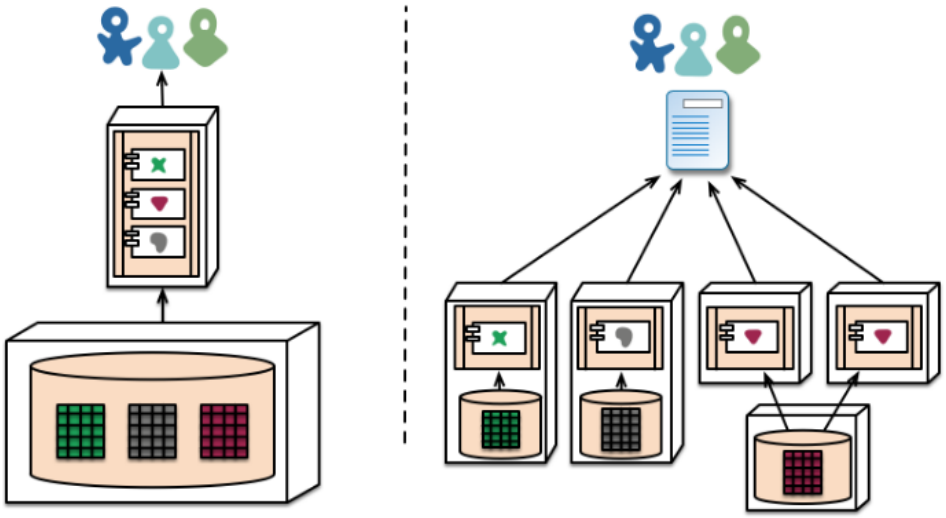
Bağlantı hattı HTTP üzerine kurgulu çağrılardan oluşabilir veya RabbitMQ, ZeroMQ gibi hafif kuyruk hatlarıyla bileşenler arası iletişim gerçekleştirilebilir.

Monolitik mimarilerde iletişim kapsüle edilmiş yapı içinde bir bütün halinde sağlanmaktadır ve bu sebeple monolitik bir uygulamanın mikroservis altyapıya dönüştürülmesindeki en kritik noktalardan biri iletişim hattıdır [6].

2.3.5 Merkezi Olmayan Veri Yönetimi

Veri yönetiminin merkezi bir yaklaşımdan merkezi olmayan bir yaklaşıma dönüşümü veya kurgusu birçok şekilde olabilir. Büyük şirketlerde entegrasyon yapılırken en büyük sorunlardan en başta gelenlerinden bazıları veri ve verinin tutulduğu ortamlar.

Monolitik mimarilerde genellikle tek ve merkeziyetçi bir veri saklama yaklaşımı varken, mikroservis mimari her servisin verisinin sadece ilgili servise ait olması gerektiğini vurgulamaktadır. Bu sayede, her servis için farklı veri saklama teknolojileri kullanılabileceği gibi bir servis yeniden yazılırken veri aktarımı ve kapsamı sadece o servisle alakalı olacağından daha problemsiz bir işlem gibi görülmektedir [6]. Bir servisin başka bir servis verisine ihtiyaç duyması halinde onunda iletişim hattı üzerinden o servislekonuşmalı ve o veriye erişebilmelidir. Veri tabanı seviyesinde bir servisin kendisine ait olmayan veriye herhangi bir erişimi olmamalıdır. Şekil 2.6'da monolitik bir uygulama ve mikroservis uygulama ile ilgili veri yönetimi özetlenmektedir.

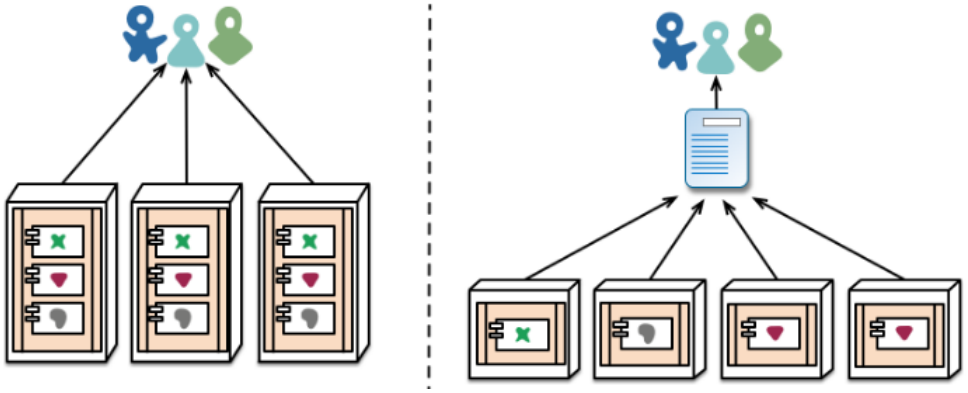


Şekil 2. 6 Çapraz işlevsel takımlar ve servis kapsülleri

2.3.6 Altyapısal Otomasyon

Altyapı otomasyon teknikleri son birkaç yılda çok gelişme göstermiştir. Bulut teknolojilerin gelişimi ve özellikle AWS'nin pazardaki yeri, mikroservis altyapılarının hazırlanmasını, kurulumunu ve çalıştırılmasındaki operasyonel karmaşıklığı azaltmaktadır.

Mikroservislerle üretilen ürün veya sistemlerin çoğu, Tümleşik Dağıtım (Continuos Delivery) tecrübesi olan ekipler tarafından inşa edilmektedir. Bunun öncü süreci Sürekli Entegrasyon'dur (Continuous Integration). Bu yolla yazılım üreten ekipler, altyapı otomasyon tekniklerini kapsamlı bir şekilde kullanmaktadırlar [6], [9].



Şekil 2. 7 Monolitik ve mikroservis altyapısal otomasyonu

Tümleşik Dağıtım'ın (Continuos Delivery) birkaç temel özelliğe dikkat çekmek gerekirse, yazılımın canlı (production) ortama çıkışını tüm fazları otomatize ederek olağan hale getirmeyi amaçlar. Ürün canlı ortama çıkmadan önce tüm otomatik testler çalıştırılır ve işlemlerin başarı ile tamamlanmasının ardından ürün canlı ortama neredeyse arkasında herhangi bir kuşku bırakmayacak şekilde çıkmış olur. Yazılımın çalıştığından mümkün olduğunca fazla güvenilmesini istendiğinden, bu nedenle çok sayıda otomatik test gerçekleştirilir.

2.3.7 Altyapısal Hatalara Toleranslı Tasarım

Mikroservis mimaride her servisi bir bileşen olarak kullanmanın bir sonucu olarak, uygulamalar, servisleri başarısızlığını -çökme erişim problemi vb.- tolere edebilecekleri şekilde tasarlanmalıdır. Bu durum monolitik bir tasarıma kıyasla bir dezavantajdır, çünkü bu işlemler karmaşıklığı da beraberinde getirmektedir [6].

Tasarlanan yapıda oluşabilecek herhangi bir servis kaybı tolere edilebilecek şekilde alarm mekanizmaları kurulmalı ve çalışmayan bileşenler için otomatik cevaplar oluşturularak sistemi kullanan müşterilere daha anlamlı bilgiler verilmelidir. Bu hem müşteri memnuniyetinde daha az hasara yol açar hem de oluşan bir probleme anında otomatik veya manuel olarak müdehaleyi kolaylaştırır.

2.3.8 Evrimsel Tasarım

Mikroservis uygulayıcıları genellikle evrimsel bir tasarım geçmişinden gelirler ve servis ayrıştırmayı, uygulama geliştiricilerinin değişiklikleri yavaşlatmadan uygulamadaki değişiklikleri kontrol etmesini sağlamak için bir başka araç olarak görürler. Değişim kontrolü, mutlaka değişimin azaltılması anlamına gelmez. Doğru tutum ve araçlarla, yazılımda sıklıkla, hızlı ve iyi kontrol edilen değişiklikler yapılabilmektedir.

Bir yazılımı bileşenler haline getirmeye çalışıldığında, parçaların nasıl bölüneceği kararıyla karşı karşıya kalınmaktadır. Bu noktada bir bileşenin temel özelliği bağımsız olarak değiştirilebilirliği veya bağımsız olarak bir üst seviyeye taşınabilirliği ile alakalıdır [6]. Mikroservis bir alt yapı bu değerler göz önünde bulundurularak tasarlanır.

BÖLÜM 3

YAZILIMDA KALİTE VE ÖLÇME

Yazılım kalitesi, müşteri taraflı kalite ve yazılımcı taraflı kalite olmak üzere, iki farklı bakış açışıyla ele alınabilir. Her müşteri doğal olarak yazılımların kolay kullanılabilir, hatasız, tüm isteklerini karşılayan ve yeterli performansa sahip olmasını istemektedir. Buna karşılık yazılım üreticileri ise geliştirme ve bakım maliyetlerinin düşük ve üretilen yazılımın parçalarının bir sonraki projelerinde de kullanılabilir olmasını istemektedirler. Yazılım üreticileri tarafında ise durum biraz daha farklıdır, düşük maliyetli, modüler, işlevsel ve bakımı yapılabilir ürünler ortaya koymayı hedeflerler.

Bu bölümde kalite standartları ve ölçme amacıyla yapılan çalışmalara yer verilmektedir. Standartlar, ölçme işlemi ve literatürdeki metrik kümeleri kısaca açıklanmıştır.

3.1 Kalite Standartları ve Kalite

ISO 9126 yazılım ürünlerinin kalitesi anlatan ve sınıflandıran uluslar arası bir standarttır. ISO 9126' ya göre kalite sınıfları ve alt sınıfları aşağıda belirtilmiştir [11].

İşlevsellik: Yazılımın ihtiyaçları karşılama yetkinliği olarak değerlendirilir. Uygunluk, birlikte çalışabilirlik ve güvenlik konuları bu kategori altında incelenmektedir.

Güvenilirlik: Yazılımın beklendiği şekilde çalışma durumunu inceler. Olgunluk, hatalara karşı toleransı olma ve geri kurtarma konuları bu kategori altında incelenmektedir.

Kullanılabilirlik: Yazılımın kullanım kolaylığı açısından incelenen yetenekleri olarak tanımlanmaktadır. Öğrenebilme, anlaşılabilirlik, işletilebilirlik ve kullanıcı etkileşimi konuları bu kategori altında incelenmektedir.

Verimlilik: Yazılımın ihtiyaç duyulan ölçüde yeterli performansla çalışabilme becerisi olarak tanımlanmaktadır. Zaman ve kaynak kullanımı konuları bu kategori altında incelenmektedir.

Bakım yapılabilirlik: Yazılımın değişiklik veya iyileştirme isteklerine adaptasyon yeteneği olarak tanımlanmaktadır. Değiştirilebilirlik, analiz edilebilirlik, test edilebilirlik bu kategori altında incelenmektedir.

Taşınabilirlik: Yazılımın farklı çalışma ortamlarına uyum sağlayabilme yeteneği olarak tanımlanmaktadır. Adaptasyon yeteneği, yüklenebilirlik özellikleri, ortam değiştirme imkânı ve diğer yazılımlarla uyum konuları bu kategori altında incelenmektedir.

Toplam kalite yönetiminin, üretim bantlarında kullanılmaya başlamasıyla birlikte, endüstriyel ürünlerde kalitenin sistematik bir biçimde arttığı çeşitli tecrübelerle saptanmıştır. Birçok şirket toplam kalite sistematiklerini kullanmaktadır. Toplam kalite yönetimi süreçlerin etkin yönetimini amaçlamaktadır. Süreç kısaca, girdileri çıktılara dönüştüren işlemler kümesi olarak tanımlanabilir [12].

Toplam kalite disiplinindeki temel esas, kaliteli ürünlerin ancak iyi tanımlanmış olgun süreçler sonucunda çıktığıdır. Mevcut süreçlerin sürekli geliştirilmesi ve iyileştirilmesi ile daha kaliteli ürünlerin ortaya çıkartılması hedeflenmektedir. Günümüzde toplam kalite prensiplerini temel alan SPICE, ISO ve CMMI gibi uluslar arası standartlar yazılım şirketleri tarafından süreç çalışmalarında sıklıkla kullanılmaktadır [13].

Bir yazılım geliştirme süreci mercek altına alındığında; müşteri isterleri, tasarım dokümanları, yazılım kodu, test sonuçları ve benzer türlerde çıktılar oluşacaktır. Süreç esnasında toplanan ölçümler, sürecin değerlendirilebilmesi ve kontrol edilebilmesi için kullanılan en önemli girdilerdir. Ölçme ise kavram olarak varlıkların özelliklerinin sayısallaştırılması olarak tanımlanabilir. Bu noktada karşımıza bir yazılımda neleri sayılaştırabileceğimiz soruları çıkmaktadır.

3.2 Metrikler

Kalite kavramının en somut göstergelerinden biri ölçüm sonuçları, ölçüm sonuçlarını meydana getiren kriterler ise metriklerdir. Süre gelen araştırmalar sonucunda nesneye

dayalı altyapılarda analiz, süreçsel iyileştirmeler ve yazılım sistemlerinin ölçeklenmesi için metrikler tanımlanmıştır.

Yazılımın kodlama ve tasarımına yönelik metrikler çeşitli açılardan kategorize edildiğinde statik ve dinamik iki tür metrik sınıfı vardır. Statik metrikler yazılım çalıştırılmadan yazılımın yapısıyla ilgili belgelerden elde edilir. Dinamik metrikler ise yazılım çalışması sırasında toplanan verilere dayanmaktadır.

Tez çalışmasında mikroservis mimariler için üzerinde çalışılan ve düşünülen metriklere benzeyen boyut, karmaşıklık ve ilişki bağımlılıklarıyla daha çok ön plana çıkan nesneye dayalı metrikler ve açıklamaları aşağıda kısaca özetlenmiştir [14], [15].

Cyclomatic Karmaşıklık - Cyclomatic Complexity (CC) : Geliştirilen kodların karmaşasının ölçülmesi olarak ifade edilir. Ölçüm yapılırken akış diyagramındaki bağımsız bölümler ve karar yapıları göz önünde bulundurulur. Ne kadar çok karar yapısı bulunuyorsa, kod karmaşası o kadar yüksek demektir. Kod karmaşasının arttığı durumlarda üzerinde çalışılan projenin sürdürülebilir olması güçleşir, hata ile karşılaşma oranında artış gözlemlenir. Bu nedenle kod karmaşasının minumumda tutulması her zaman için önerilir.

Sınıfın Ağırlıklı Metot Sayısı - Weighted Methods per Class (WMC) : Bir sınıfın tüm metotlarının karmaşıklığının toplamıdır. Metot sayısı çok olan taban sınıflar, çocuk düğümlerde daha çok etki bırakırlar. Tekrar kullanılabilirliği olumsuz etkiler.

Nesne Sınıfları Arasındaki Bağımlılık – Coupling Between Object Classes (CBO): Sınıfın bağımlı olduğu sınıf sayısıdır. Bir sınıf içinde nitelik ya da metotlar diğer sınıfta kullanılıyorsa ve sınıflar arasında kalıtım yoksa iki sınıf arasında bağımlılık olduğu kabul edilmiştir. Düşük tutulması gereken temel metriklerden biridir.

Sınıf Arayüz Boyutu - Class Interface Size (CIS) : Sınıfın açık (public) metotlarının sayısıdır ve yazılımın mesajlaşma özelliği hakkında fikir verir.

Doğrudan Sınıf Bağımlılığı- Direct Class Coupling (DCC) : Bir sınıfı parametre olarak kabul eden sınıfların sayısı ile bu sınıfı nitelik değişkeni olarak barındıran sınıfların sayısının toplamıdır. Sınıf bağımlılıkları için kritik bir metriktir.

Metot Sayısı - Number of Methods (NOM) : Sınıfta tanımlı metot sayısı olarak değerlendirilmektedir. Sınıfın metot sayısı sınıfın karmaşıklığıyla ilgili bilgi verir.

3.3 Yardımcı Araçlar

Birçok metriği manuel olarak analiz etmek çok uzun sürmektedir. Uzun sürmese dahi çok sık analiz gerekliliği metriklerin otomasyonel bir sistem ile analiz edilmesi gerekliliğini doğurmuştur. Bu kapsamda popüler araçlardan FindBugs, PMD, Checkstyle ve Coverity analiz araçlarına bir göz atalım.

FindBugs [16], açık kaynaklı bir statik kod analiz aracıdır. Java kodu üzerinde çalışmaktadır. Eclipse, netbeans, jboss gibi IDE'lerde ve kalite entegrasyon uygulamalarına eklentileri mevcuttur.

PMD [17] ise başka bir statik kod analiz aracıdır. Java kodları üzerinde çalışan bu program mevcut kod üzerinde otomatik analizler yaparak olası kusurları; kullanılmayan, tekrarlanmış ve gereksiz kod parçalarını hızlıca keşfedebilmektedir.

CheckStyle programı [18] yazılımın yapısından çok formatı ile ilgilenen açık kaynak kodlu yazılım aracıdır. Java kodlarının üzerinde format analizi yaparak geliştiricilerin kod yazım standartlarına uyumlu çalışmalarına yardımcı olmaktadır. Kurumun kendi yazım standartlarını da oluşturabileceği bu yazılımda, aynı zamanda genel kabul görmüş kimi uluslararası yazım standartları da mevcuttur.

Coverity [19], Java'nın yanı C++ ve C program kodları üzerinden çalışabilen ticari ve oldukça kapsamlı bir kod analiz aracıdır. Otomatik düzeltme desteği de sunmaktadır. Bu programın diğer bir önemli özelliği de, statik analizin yanı sıra çalışma esnasında dinamik analizleri de yapıp raporlayabilmesidir.

BÖLÜM 4

MİKROSERVİS TABANLI UYGULAMALAR İÇİN METRİKLER VE PRATİKLER

Bu bölümde mikroservis mimari stili üzerine geliştirilen uygulamaların analiz edebilmesi için öne sürdüğümüz metrikler ve pratikler ele alınmaktadır.

Metrik ve pratikler üç kategori altında incelenmektedir. Bunlar, servis boyutunu ölçümlemeye yönelik metrikler, servislerarası ilişkileri düzenleme amacındaki bulgular ve metrikler, uygulanmasından kaçınılması gerekilen pratiklerdir. Metrik ve pratikler herhangi bir yazılım dili veya teknolojisine bağlı olmamakla birlikte, mikroservis mimarilerde yaşanabilecek olumlu veya olumsuz senaryolar düşünülerek öne sürülmüştür.

4.1 Servis Boyutunu Ölçümleme

Mikroservislerde gri alanlardan biri "mikro" olarak tanımlanan kavramın ne büyüklükte olduğunun tam anlamıyla bilinmemesidir. Bununla ilgili çeşitli öneriler olmakla birlikte, ekrana sığmalı, tek bakışta kod okunabilmeli gibi yaklaşımlarla sınırlıdır.

Yazılım dünyasında yazılım boyutunu ölçümlemek için çeşitli metric ve yaklaşımlar mevcuttur. Kaynak Kod Satır Sayısı (Source Lines Of Code - SLOC) ölçümleme, Mantıksal Kod Satır Sayısı (Logical Lines Of Code - LLOC) ölçümleme ve Use Case tabanlı yöntemlerle işlevsellik odaklı ölçümlemeler gibi yöntemler bunlardan sadece birkaçının ismidir [20], [21].

Servis boyutu ölçümleme işleminde, bir mikroservisin diğer mikroservislerle iletişime girebileceği noktalar ölçüm kriteri olarak belirlenir. **Kaynak** (resource), bir URI ile tanımlı network veri nesnesini ifade ederken, **İstemci**(client) ise bir istek göndermek

için hazırlanmış bir programı veya olguyu ifade eder [22]. Ölçüm, Kaynak ve İstemci Sayıları dikkate alınarak yapılmaktadır.

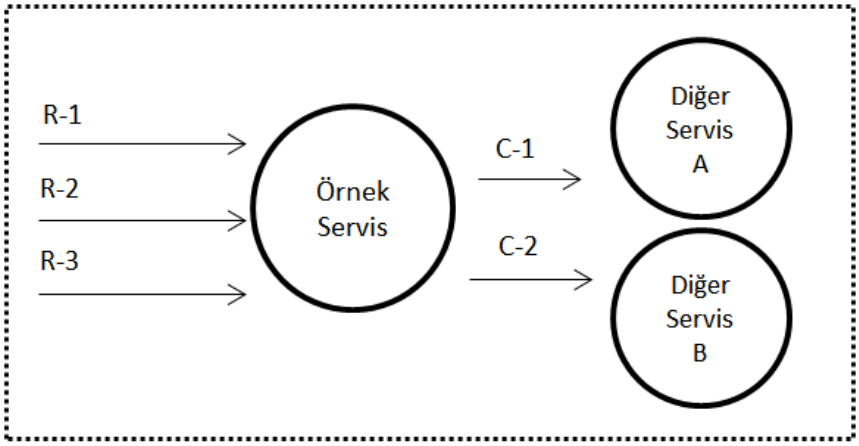
Yöntem temelde, Sınıf Arayüz Boyutu (CIS) metriğinin kıstasları ile görünüm olarak benzerlik gösterse de, çalışmamızda servisler arası iletişimde kullanılan fonksiyonaliteler servis boyutunu belirlemek amacıyla kullanılmaktadır.

Kaynak Sayısı – Resource Count (RC) : Servisin sahip olduğu istek kabul etme noktalarının sayısı bu metriğin değerini verir. Değerin yüksek olması servisin sorumluluk kriterini de belirler. Ne kadar büyük servis o kadar büyük sorumluluk demektir; fakat önem derecesi ile karıştırılmamalıdır.

İstemci Sayısı – Client Count (CC) : Bir servisin diğer servislerle yapacağı istekler için sahip olduğu istemcilerin sayısı metriğin ölçüm değerini verir. Değerin yüksek olması servisin diğer servislere muhtaçlık seviyesi ile doğru orantılıdır.

Şekil 4.1'e göre "Örnek Servis" mikroservisi, R-1, R-2 ve R-2 kaynaklarına sahiptir. Aynı zamanda C-1 ve C-2 istemcileri ile de farklı servislere istek göndermektedir. Tanımlanan metrik kriterlerine göre "Örnek Servis" 3 adet kaynak, 2 adet ise istemci barındırmaktadır.

Örnek Servis'in boyutunu S olarak ifade edecek olursak, ***S = RC + CC*** olarak kurgulanacak bir formül ile servis boyutu **S = 3 + 2**, yani 5 olarak ölçümlenecektir.



Şekil 4. 1 Örnek Servis için kaynaklar ve istemciler

Şekil 4.1'de Örnek Servis'in istekte bulunduğu Diğer Servis A ve Diğer Servis B herhangi bir şekilde farklı servisler ile iletişime geçmiyorlar ise, sadece tek bir isteği kabul

ettiklerinden kaynak sayıları 1 olarak belirlenecek ve her bir servisin boyutu **S = 1 + 0**, yani 1 olarak ölçümlenecektir.

Servislerin diğer servislerle arasındaki boyut farklılıklarının servislerin tasarımını gözden geçirme konusunda fikir verici olacağı düşünülmektedir.

4.2 Servisler Arası İletişim Uygunluğunu Ölçümleme

Mikroservis mimarilerde her bir servisin diğer servislerle olan iletişimi en kritik konulardan biridir. İletişim dışında kalan bir servisin hizmet vermesi mümkün değildir. Bu tarz durumlar network arayüz hatalarından, servislerde tanımlanan URL hatalarından veya birçok geliştirici ile yapılan geliştirmelerde geliştiriciler arası kültürel ve geliştirme kalitesi farklılıklarından kaynaklanıyor olabilir.

Servis sayısının yüzler seviyelerine çıktığı durumlarda, bir servisi geliştiren kişi veya ekipler çok farklı lokasyonlarda çalışıyor olabilirler.

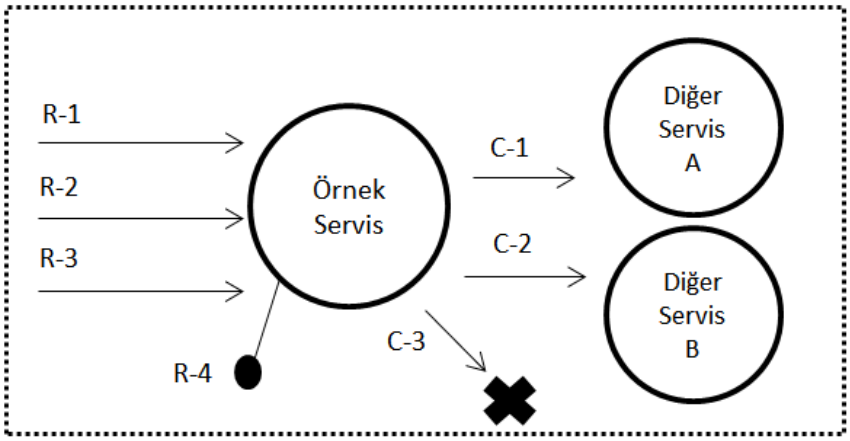
Bu bölümdeki ölçümlemelerde, servislerin birbirlerine yaptıkları çağrıların doğru ve uyumlu olup olmadığı incelenmektedir. Kullanılmayan, diğer bir deyişle atıl kalan kaynak ve istemciler bu kategorideki çalışmanın odak noktalarıdır.

Kullanılmayan Kaynak Sayısı – Unused Resource Count (URC) : Bir serviste var olan, uygulama bütünü kapsamında başka hiçbir servis tarafından istekte bulunulmayan kaynak sayısıdır. Bu metrik kavramsal olarak, nesneye dayalı bir yazılımda, sınıf içinde kullanılmayan ve kod içinde yer kaplayan bir method denginde düşünülebilir.

Erişilemeyen İstemci Sayısı – Unreachable Client Count (UCC) : Bir serviste var olan, uygulama bütünü kapsamında başka hiçbir servisteki kaynakla eşleşmeyen istemcilerin sayısıdır. Nesneye dayalı bir yazılımda, bir sınıfın başka bir sınıfın var olmayan bir methoduna ait çağrıya sahip olması olarak düşünülebilir.

Şekil 4.2'de kullanılmayan kaynak ve istemcileri de içeren bir mimari bulunmaktadır. Örnek Servis **R-4** olarak belirtilmiş bir kaynağa sahiptir. Bu kaynak hiçbir mikroservis tarafından kullanılmamaktadır. Bu sebeple Örnek Servisi'nin Kullanılmayan Kaynak Sayısı (**URC**) 1 olarak ölçümlenir. Diğer yandan Örnek Servis **C-3** adında diğer hiçbir

servisle ilişkisi olmayan istemciye sahiptir. Bu sepeple Örnek Servis'in Kullanılmayan İstemci Sayısı (**UCC**) 1 olarak ölçümlenir.



Şekil 4. 2 Örnek Servis için kullanılmayan kaynak ve istemciler

4.3 Servis Keşfedilebilirliğini Ölçümleme

Mikroservis mimarilerde, servisler ölçeklenebilir olmalıdır ve her an işlem bakımından sonlandırılıp tekrardan sürece hızlıca dâhil olabilmelidirler. Bu amaçla Consul, Eureka gibi servis keşfi sağlayan araçlar geliştirmiştir. Temelde amaç, sürece dâhil olmak isteyen her servisin kendini servis keşfedicilere kaydetmesi ve servisi kullanacak diğer servislerin o servisi keşfediciden öğrenmeleridir. Bu sayede aynı zamanda hiçbir servisin diğer servislerin hangi IP veya alan isminde var olduğunu tutması gerekmeyecektir. Sadece kayıtları tutan bileşene sorması yeterlidir.

Bu bölümde servislerin keşfi yaklaşımını devre dışı bırakan, mikroservis prensiplerine aykırılık teşkil eden veya olumsuz etkileyen URI tabanlı iki kötü pratik tanımlanmıştır.

Sabit URI – Static URI (SURI) : Bir servisin diğer bir servise direk olarak kod içinde veya kod konfigürasyonunda tanımlı sabit bir IP adresi ile erişiyor olması olarak tanımlanır. Servis ölçeklemesi esnasında, ilgili servisin farklı bir IP'de çalışmaya başlaması durumunda, tanımlı sabit IP erişilemez bir uç nokta haline dönüşür ve IP adresi statik olarak belirtilen servisle olan iletişim kesilir. Yazılımın güvenilirliği açısından bu tür pratiklerle geliştirme yapmak riskli bir davranıştır.

Uzun URI – Long URI (LURI) : Bir servisin sahip olduğu kaynak URI'sinin parametrik kısımları hariç 50 karakterden uzun olması durumudur. Bu değer yazılı ekibinin tercihi

doğrultusunda esnetilebilir. API iletişiminde anlaşılamayan veya mantıksal bir kurguya sahip olmayan URI'lerin sektörel deneyimlere dayanarak kullanım anlamında yazılımcıyı zorlamakta ve hata durumunda loglar üzerinde hata ayıklamayı zor hale getirmektedir.

BÖLÜM 5

STATİK KOD ANALİZ ARACI

Bir önceki bölümde tanımlanan metrik ve pratiklerin otomasyonel analizi için kaynak kod üzerinde çalışan bir araç geliştirilmiştir. Bu bölümde aracın çalışma yöntemi ve çıktılarıyla ilgili bilgiler yer almaktadır.

5.1 Statik Kod Analizi Nedir

Statik kod analizi, bir yazılımın içerdiği hataların yazılım çalıştırılmadan tespit edilmesi ve sahip olduğu çeşitli özelliklerin elde edilmesi amacıyla yapılan çalışmalar bütünüdür.

Yazılımların statik olarak analiz edilmeleri gözden geçirmeler şeklinde yapılabileceği gibi statik kod analiz araçlarının kullanıma alınmasıyla otomatik olarak da yapılabilmektedir. Her iki yöntemin de kendi içinde avantaj ve dezavantajları bulunmaktadır. Bununla birlikte, insan faktörüne bağlı olarak gözden geçirmelerde zamanla oluşan dikkat dağılması hataların gözden kaçırılmasına neden olabilmektedir. Otomatik statik kod analizinde ise yazılımda bulunan problemler çok hızlı bir biçimde tespit edilebilmektedir. Ayrıca yazılım hakkında istenen pek çok bilgi ve analiz sonuçları kullanılan araçlar yardımıyla metinsel ve grafiksel olarak elde edilebilmektedir. Sonuç olarak, otomatikleştirilmiş statik kod analizinin ölçeklenebilir olması, tutarlı yapısı ve maliyet açısından uygun olması sebebiyle kaynak kod gözden geçirme yöntemine göre öne çıkmaktadır [23].

Statik kod analizi ile hata ve güvenlik açıkları tespiti, kaynak kod metrik hesabı, mimari analiz, kodlama standartlarına uygunluk ve tersine mühendislik çalışmaları gibi birçok işlem gerçekleştirilebilir [23].

5.2 Mikroservis Statik Kod Analiz Aracı (MISKAA)

Araç Java dilinde geliştirilmiştir ve Java dilinde yazılmış kodu girdi olarak alıp tanımlı metriklerle ilgili rapor üretmektedir. Aracı diğer araçlardan ayıran özelliklerden biri birden fazla projeyi aynı anda çapraz olarak analiz ediyor olabilmesidir. Bölüm 3.3'te diğer analiz araçlarıyla ilgili bilgiler yer almaktadır. Çalışma kapsamında gerçeklenen araç **MISKAA** (Mikroservis Statik Kod Analiz Aracı) olarak adlandırılmıştır.

5.2.1 Proje Tanımı Konfigürasyon Dosyası Servis Tanımları

MISKAA, dağıtık halde GIT, SVN gibi versiyonlu kaynak kod saklama alanlarında depolanan mikroservislerin bir arada analiz edilebilmesini sağlayan konfigürasyon dosyasını çalıştırılırken parametre olarak almaktadır. Bu dosya YAML formatında oluşturulmuş **Proje Tanımı Konfigürasyon Dosyası** olarak adlandırılan dosyadır.

Konfigürasyon sayesinde yeni servisler bu dosyaya tanımlanarak analiz kapsamına alınır. Konfigürasyon tanımları kritiktir; çünkü tanımı unutulan bir mikroservis için çıktılar tez kapsamında bahsedilen metrikler dâhilinde olması gerekenden farklı sonuçlar oluşmasına sebebiyet verecektir.

Dosya formatı Çizelge 5.1'de örneklerle paylaşılmıştır. Uygulama bütünü kapsamındaki tüm servisler bu dosyaya tanımlanmalıdır.

Çizelge 5. 1 Proje Tanımı Konfigürasyon Dosyası YML formatı

```
projects:
- name: Merhaba
  location: repo//projects//merhaba//src
- name: Cumle
  location: repo//projects//cumle//src

…
```

5.2.2 Servis Analizi

MISKAA, JavaParser kütüphanesinden yararlanılarak yazılmış bir araçtır. JavaParser kütüphanesi, JAVA dilinde yazılan kodun Soyut Sözdizim Ağacı (Abstract Syntax Tree - AST) olarak adlandırılan yapısını dolaşabileceğiniz(visit) bir alt yapı sunmaktadır. MISKAA oluşturulan bu ağaçtan yararlanarak kaynak hizmeti için [24] Jersey JAX-RS

spesifikasyonunda tanımlı anotasyonlar, İstemci hizmeti için de açık kaynak kodlu Feign kütüphanesinde tanımlı anotasyonları filtreleyerek indirgenmiş yeni bir ağaç oluşturur. Daha somut bir deyişle, kodun tamamından sadece mikroservislerin birbirleriyle iletişime geçtiği kısımları alır ve analiz edilebilir bir şablon sunar.

JAVA Jersey [24] kütüphanesi kapsamında *@PATH, @GET, @POST, @PUT, @DELETE, @PATCH* anotasyonları ayrıştırılır ve servise gelen istekleri kabul edecek Kaynak listesi *.java dosyasına ait bilgilerle oluşturulur. Çizelge 5.2'de belirtilen kod parçasındaki kaynak, *[project : Merhaba, httpMethod : GET , URL : /mesaj/merhaba …]* formatın bir yapıda Kaynaklar listesinde tutulmaktadır. JAVA dilinde 1 sınıf N method içerir mantığıyla yukarıdan aşağıya kırılımlı Kaynak URL değeri oluşmaktadır.

Çizelge 5. 2 Jersey GET Kaynak tanımı

```
@Path("/mesaj")
public class MerhabaServis {

    @GET
    @Path("/merhaba")
    public Response getMesaj() {

        String cikti = "Yıldız Teknik Üniversitesi'nden Merhaba!";
        return Response.status(200).entity(cikti).build();

    }
}
```

Feign [25] kütüphanesi kapsamında *@RequestLine* anotasyonu ayrıştırılır ve servisten yapılacak istekleri oluşturun İstemci listesi *.java dosyasına ait bilgilerle oluşturulur. Çizelge 5.2'de paylaşılan kod parçasında bir servisin diğer bir servise yapacağı İstemci tanımlanmıştır. Analiz aracı bu kodu, *[project : Merhaba, httpMethod : POST, URL : /cumle/{cumle} …]* formatında bir yapıda İstemciler listesinde tutmaktadır.

Her servis bu şekilde tekil olarak analiz edilir ve tüm analiz tamamlandıktan sonra servisler arası eşleştirmeler yapılır. Tanımlı metrikler üzerinden boyutlandırma, servisler arası iletişim ve pratikler bakımından değerlendilerek rapor oluşturulur.

Çizelge 5. 3 Feign POST İstemci tanımı

```
interface CumleClient {

  @RequestLine("POST /cumle/{cumle}")
  String getCumle(@Param("cumle") String cumle);

}
```

5.2.3 Sonuç Raporları

MISKAA analiz sonuçları, sonuçlara ait JAVA nesnesi JSON formatına çevrilerek konsola ve çalıştırıldığı güne ait bilgileri içeren formattaki *"YYYY-MM-DD-1.miskaa"* dosyasına yazılır. Çıktı formatı Çizelge 5.4'te paylaşılmıştır. Çıktı bulgunun dosyadaki yerini gösteren detaylı sonuçların yer aldığı "results" kısmı ve özet metrik sonuçlarının yer aldığı "summary" kısmı bulunmaktadır.

Çizelge 5. 4 Örnek Sonuç Raporu

```
{
 "results" : [ {
  "project" : "creditcardservice",
  "detectionType" : "UNUSED_RESOURCE",
  "detectionLevel" : "ERROR",
  "location" : "Optional[(line 24,col 5)-(line 28,col 5)]",
  "file" : "\\resources\\CreditCardResource.java"
 }, {
  "project" : "ticketservice",
  "detectionType" : "STATIC_URI",
  "detectionLevel" : "ERROR",
  "location" : "Optional[(line 29,col 5)-(line 30,col 50)]",
  "file" : "\\client\\ExternalServices.java"
 }
 ...
]
"summary" : [ {
    "serviceName" : "creditcardservice",
    "detectionType" : "UNUSED_RESOURCE",
    "detectionLevel" : "ERROR",
    "count" : 3
  }, {
  "serviceName" : "ticketservice",
  "detectionType" : "STATIC_URI",
  "detectionLevel" : "ERROR",
  "count" : 1
 }
 ...
] }
```

BÖLÜM 6

BİLET SATIŞI ÖRNEK UYGULAMASI

Bilet Satış Uygulaması, tanımlanan metrik ve pratiklerin MISKAA aracı ile analizi için mikroservis tabanlı geliştirilen örnek bir uygulamadır.

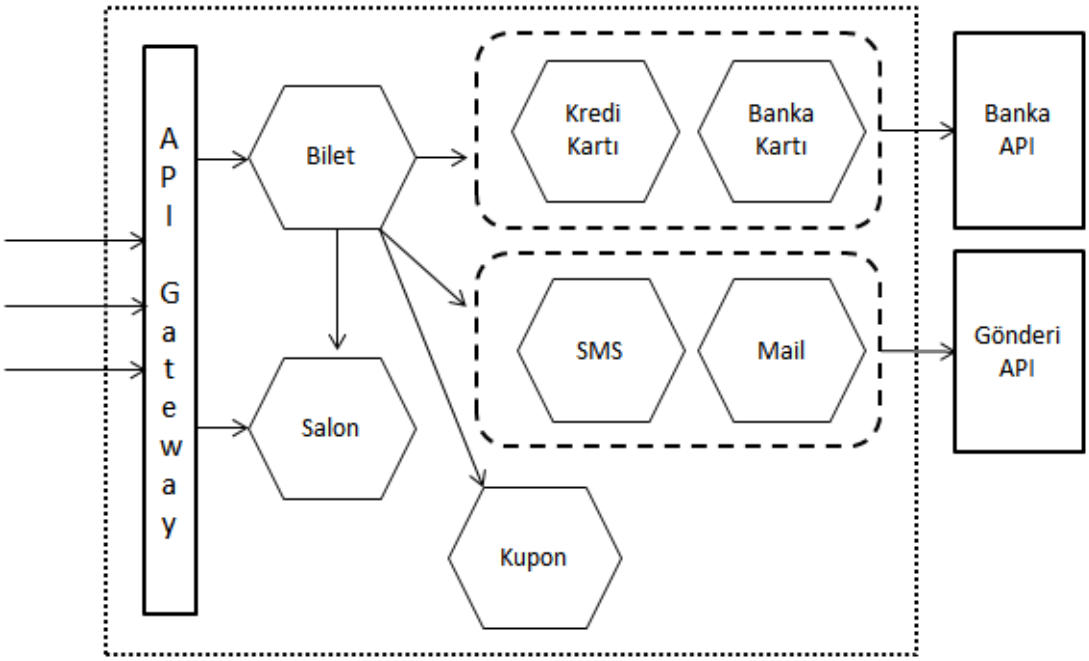
Uygulamanın geliştirilmesi esnasında bir senaryo esasına dayanarak kasıtlı olarak yapılan hatalar ve senaryo kapsamında tanımlı mikroservis ilişkileri bulunmaktadır.

Uygulama API Gateway kalıbına uygun geliştirilmiştir. Uygulama dâhilindeki mikroservisler; Salon, Bilet, Kredi Kartı, Banka Kartı, Kupon, SMS ve Mail servisleridir. Şekil 6.1'de uygulama mimarisi şematize edilmiştir. Servislere gelen istekler API Gateway üzerinden dağıtılmaktadır. Dış dünyaya çıkış bağlantıları için Banka API ve Gönderi API erişimleri bulunmaktadır.

Servis API'leri Richardson Olgunluk Seviyesi-2 düzeyinde geliştirilmiştir. Servis URL'lerinde (:id) gibi değişkenler, o değerin parametrik olduğunu belirtmektedir. Servisler arası keşif(discovery) altyapısının olduğu varsayılmıştır.

6.1 Mikroservisler

Bilet Satış Uygulaması'ndaki her servis bu bölümde incelenmiş ve tasarım senaryolarıyle birlikte API'leri tanımlanmıştır. Senaryo gereği kasıtlı yapılan hatalar işaretlidir. Sonuçlar bu senaryo ile karşılaştırılacaktır.



Şekil 6. 1 Bilet Satış Uygulaması servis mimarisi

6.1.1 Salon Servisi

Sinema veya tiyatro salonlarında olan var olan seanslarla ilgili bilgilerden ve işlevselliklerden sorumlu servistir.

Kaynaklar

- GET /salonservisi/salonlar
- GET /salonservisi/salonlar/:id
- GET /salonservisi /salonlar/sehirler/:cid
- PUT /salonservisi /salonlar/:id /koltuklar/:sidGET /salonservisi/salonlar/sehirler/:cid:/ bolgeler/:bid/kategoriler/:caid/zaman/baslangic/:basGunu/bitis/:bitGunu/... (**LongURI**)

6.1.2 Bilet Servisi

Biletleme işlemleri için tasarlanan servistir. Sistemdeki tüm servislerle ilişki halindedir.

Kaynaklar

- GET /biletservisi/biletler

- GET /biletservisi/biletler/:id
- GET /biletservisi/ biletler/seats/:id
- POST /biletservisi/biletler
- PUT /biletservisi/biletler
- DELETE /biletservisi/biletler/:id

İstemciler

- PUT / salonservisi/salonlar/:id/koltuklar/:sid
- POST /kredikartiservisi/kartlar
- POST /bankakartiservisi/kartlar
- PUT /kuponservisi/kuponlar
- POST /smsservisi/smsler
- POST /pushbildirimservisi/bildirimler (**Erişilemez**)
- POST http://10.11.12.13/mailservisi/mailler (**Statik URI**)

6.1.3 Kredi Kartı Servisi

Kredi Kartı servisi, kredi kartı ile yapılan işlemlerden sorumlu servistir. Kendine ait veri sahipliği olduğu bir veritabanı yoktur. Bankanın API'leri ile konuşmaktadır. Banka ile uygulama arasında yarı geçirgen bir katman rolünü üstlenir.

Kaynaklar

- POST /kredikartiservisi/kartlar
- POST /kedikartiservisi/visa (**Kullanılmayan Kaynak**)
- POST /kredikartiservisi/master (**Kullanılmayan Kaynak**)
- POST /kredikartiservisi/americanexpress (**Kullanılmayan Kaynak**)

6.1.4 Banka Kartı Servisi

Banka Kartı Servisi, banka kartı ile yapılan işlemlerden sorumlu servistir. Kendine ait veri sahipliği olduğu bir veritabanı yoktur. Bankanın API'leri ile konuşmaktadır. Banka ile uygulama arasında yarı geçirgen bir katman rolünde görevi vardır.

Kaynaklar

- POST /bankakartiservisi/kartlar

6.1.5 Kupon Servisi

Ödeme işlemlerinde promosyon kuponlarından sorumlu servistir.

Kaynaklar

- PUT /kuponservisi/coupons
- GET / kuponservisi/coupons (**Kullanılmayan Kaynak**)

6.1.6 SMS Servisi

SMS gönderimlerinden sorumlu servistir. Gönderi API'leri ile sistem arasında yarı geçirgen bir rol üstlenerek SMS gönderim işlemindeki sorumluluğunu yerine getirir. Kredi Kartı Servisi ve Banka Kartı Servisi'nden farklı olarak gönderilen SMS'ler için kendine ait veritabanı vardır.

Kaynaklar

- POST /smsservisi/smsler

6.1.7 Mail Servisi

Mail gönderiminden sorumlu servistir. Gönderi API'leri ile sistem arasında yarı geçirgen bir rol üstlenerek SMS gönderim işlemindeki sorumluluğunu yerine getirir. Kendine ait veri sahipliği olduğu bir veritabanı yoktur.

Kaynaklar

- POST /mailservisi/mailler

6.2 Servis Analizi ve Sonuçların Değerlendirilmesi

Kasıtlı olarak hatalar oluşturulan örnek uygulama analiz sonuçları beklenen ile aynı oluştuğu görülmüştür. Servislerin sahip olduğu değerler metrikler açısından doğru şekilde ölçümlenmiştir.

Analiz çıktılarını içeren Çizelge 7.1 paylaşılmıştır. Çıktılara göre Bilet Servisi'nin en diğer servislere istek gönderen tek servis olduğu görülmektedir. Diğer yandan Banka Kartı Servisi, SMS servisi ve Mail servisi sadece 1 kaynağa sahiptir ve boyutları 1 olarak ölçüm sonuçlarına yansımıştır.

Çizelge 7. 1 Bilet Satış Uygulaması analiz sonuçları

Servis	Kaynak Sayısı	İstemci Sayısı	Boyut	Kul.mayan Kaynak Sayısı	Erişilemeyen İstemci Sayısı	Sabit URI	Uzun URI
Salon	5	0	5	0	0	0	1
Bilet	6	7	13	0	1	1	0
Kredi Kartı	4	0	4	3	0	0	0
Banka Kartı	1	0	1	0	0	0	0
Kupon	2	0	2	1	0	0	0
SMS	1	0	1	0	0	0	0
Mail	1	0	1	0	0	0	0
Toplam	20	7	27	4	1	1	1

Toplamda 1 adet Uzun URI bağlantı tespit edilmiş ve raporlanmıştır. Bu durum uygulamanın işlevselliğinde herhangi bir bozukluğa sebep olmayacak olmakla birlikte,

sürekli analizler sonucunda LURI sayısının artması üzerinde durulması gereken bir konu haline dönmelidir.

Bilet Servisi farklı bir servise sabit bir IP adresi üzerinden gitmeye çalışmaktadır. İlgili servis aynı IP adresinde yaşam döngüsüne devam ettiği sürece bir problem olmayacak; fakat herhangi bir IP değişikliğinde Bilet Servisi o servisten alması gereken hizmeti alamayacağı için fonksiyonal aksaklıklar oluşacaktır. Bilet Servisi'nde sabit IP adresi ile eriştiği servisin her değişikliğinde manuel olarak güncelleme yapılması gerekmektedir. Bu durum sürekli değişken ölçekli yaşayan bir mikroservis ekosisteminde istenmeyen bir durumdur. Kod geliştirme aşamalarında herhangi bir kod kalite ölçütü uygulanmadığında bu tarz durumlarla karşı karşıya kalınması çok olasıdır.

Bilet Servisi metriklerimize göre en büyük servis olarak ölçümlenmiştir. Bu durum Biletleme Servisi'nin kod satır sayısının veya uygulama dosya boyutunun diğer servislerden daha büyük veya küçük olduğu anlamına gelmez. Tanımlanan boyut ölçüm kriterleri diğer ölçüm kriterlerinden bağımsız olarak oluşturulmaktadır.

Kredi Kartı Servisi'nde 3 ve Kupon Servisi'nde 1 olmak üzere toplamda 4 adet kullanılmayan kaynak tespit edilmiştir. İlgili kaynak tanımları kod içinde unutulmuş olabilir veya henüz diğer servisler tarafından geliştirilmesi yapılmadan sadece o servislerde versiyon yükseltilmiş olabilir.

Sadece hazırladığımız örnek uygulamanın senaryolarına bakarak bile Çizelge 7.1 'deki gibi bir analiz sonucu çıkarmamız 5-6 dakikadan daha fazla süre alacaktır. Hele ki, sürekli değişen bir kod havuzunda bunu yapmamız bir yerden sonra mümkün hale gelmeyecektir. Otomatik bir sürecin faydasını burada görebiliyoruz.

Sürekli iyileştirmenin olduğu ve kalitenin kontrol altında tutulduğu sistemlerin hata oranının azalacağı düşünülmektedir. Bu standartlaşma seviyesine ulaşmanın yolu, uygulamalar üzerinde hem otomatik, hem uzman personel kararları ile denetlenen kontrol mekanizmaları oluşturmaktan geçmektedir. Sektörel deneyimlerle gözlemlenmiştir.

BÖLÜM 7

SONUÇ VE ÖNERİLER

Hızla artan yazılım maliyetlerini azaltmak, daha kaliteli ürünler ortaya koymak ancak geliştirilen yazılımda önce kaliteyi korumak ve daha sonra kaliteyi artırmaya yönelik çalışmalarda bulunarak mümkündür. Tom DeMarco' nun ifade ettiği “Ölçemediğimiz bir şeyi kontrol edemeyiz ve iyileştiremeyiz” düşüncesi doğal olarak yazılım alanına da temas etmektedir.

Tez çalışması kapsamında bir şeyleri ölçebilmek adına yeni metrikler ve pratikler tanımlanmış, tanımlanan kriterlerin analiz edilebileceği Bilet Satış uygulaması senaryosu yapay servislerle hayata geçirilmiştir. Tanımlı metrik ve pratiklerin analizi için Java kaynak kodu üzerinde çalışan MISKAA analiz aracı geliştirilmiştir. Sonuçlar metrikler bazında bu araç ile değerlendirilmiş ve metrik bazında analiz sonuçları tartışılmıştır.

Mikroservis mimari yeni ivme kazanan bir mimari olup Amazon, Netflix gibi büyük firmalar tarafından kabul görmekle birlikte, yakın zaman dilimi için bile olsa tam anlamıyla teknik anlamda tatmin edici teknik kanıtlar ile varlığını henüz ıspatlamamıştır. Yapılan çalışmanın mikroservis mimari kültürüne fayda sağlayacağı düşünülmektedir.

Mikroservis mimarideki diğer yaklaşımlardan biri farklı yazılım dilleri ile servis ekosistemi kurmaktır. Bu açıdan bakıldığında geliştirilen metrikler için bir değişiklik olmasa bile, analiz eden MISKAA aracı farklı dil ve çatılarda istenen sonuçları üretemeyecektir. Aracın sadece Java kodunu ve belirli çatıları kabul ediyor olması bir dezavantajıdır.

Genel olarak yazılım kalitesi ölçümlenmesi planlanır ve belirli parametrelerin değerlendirilmesi düşünülürse, sadece otomasyonal statik araçlar kullanımından çok hem dinamik analiz yapan alternatifler hem de manuel kontrol yapmayı sağlayan kod gözden geçirme araçlarının kullanılması önerilmektedir.

Geliştirmeler sonucunda elde edilen raporlar dahilinde, servisler gerektiğinde daha küçük parçalara bölünebilir veya mantıksal olarak bütün halinde bulunması gereken parçalar –çok önerilmese de- kontrollü olarak birleştirilebilir.

Yazılım geliştirmede onlarca kişiyle geliştirme yapılan durumlarda otomasyonel analizler iş gücü kaybının önüne geçeceğinden, otomasyonel süreç yazılım geliştirme sürecinin her adımında önemlidir ve önerilmektedir.

KAYNAKLAR

[1] Microsoft, (2009). What is Software Architecture?, Microsoft Application Architecture Guide 2nd Edition.

[2] Wikipedia, Yazılım Mimarisi, https://tr.wikipedia.org/wiki/Yaz%C4%B1l%C4%B1m_mimarisi, 28 Nisan 2017.

[3] Gökalp G., (2016), Monolithic ve MicroService Architecture'a Genel Bir Bakış, http://www.gokhan-gokalp.com/monolithic-ve-microservice-architecturea-genel-bir-bakis, 5 Mayıs 2017.

[4] Erl, T., (2009). Service-Oriented Architecture Concept, Technology and Design, San Francisco.

[5] Code Archive Blog, SOA Nedir?, http://www.coderarchive.net/genel/soa-nedir-service-oriented-architecture/162, 2 Mayıs 2017.

[6] Lewis J. ve Fowler M., (2014), Microservices, https://martinfowler.com/articles/microservices.html, 12 Şubat 2017.

[7] Wikipedia, Enterprise Service Bus, https://en.wikipedia.org/wiki/Enterprise_service_bus, 5 Mayıs 2017.

[8] Nadareishvili I., (2016). Microservice Architecture: Aligning Principles, Practices and Culture, OReilly.

[9] Namiot D. ve Sneps-Sneppe M., (2014). "On Microservices Architecture", International Journal of Open Information Technologies, 9: 24-27.

[10] Thones J., (2015). "Microservices", IEEE Software, 1: 116.

[11] ISO 9126, (1977). Software Quality Characteristics, An overview of the ISO 9126-1 software quality model definition, with an explanation of the major characteristics, ISO.

[12] Booch G., (1991). Object Oriented Design with Applications, Benjamin Cummings, Redwood City, California.

[13] Erdemir U., Tekin U. ve Buzluca F., (2008). "Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi", 1: 2-5.

[14] Berard E., Metrics for Object-Oriented Software Engineering, http://www.ipipan.gda.pl/~marek/objects/TOA/moose.html, 9 Mayıs 2017.

[15] Jamali S.M., (2006). "Object Oriented Metrics", 1-8.

[16] FindBugs, Findbugs Analiz Aracı, http://findbugs.sourceforge.net/index.html, 2 Mayıs 2017.

[17] PMD, PMD Analiz Aracı, http://pmd.sourceforge.net, 2 Mayıs 2017.

[18] Checksytle, Checkstyle Analiz Aracı, http://checkstyle.sourceforge.net, 2 Mayıs 2017.

[19] Coverity, Coverity Analiz Aracı, http://www.coverity.com, 2 Mayıs 2017.

[20] Symons C., Abran A., Ebert C. ve Vogelezang F., (2016). Measurement of Software Size: Advances Made by the COSMIC Community, IWSM-MENSURA.

[21] Wikipedia, Software Sizing, https://en.wikipedia.org/wiki/Software_sizing, 23 Nisan 2017.

[22] RFC 2616, (1999). Hypertext Transfer Protocol HTTP/1.1, RFC.

[23] Kalay A., (2009). "Statik Kod Analizinin Yazılım Gelistirme Sürecindeki Yeri ve Yazılım Kalitesine Etkisi", 4. Ulusal Yazılım Mühendisliği Sempozyumu UYMS'09, 212.

[24] Jax RS 2.0, (2013). Java API for RESTful Services, Java.

[25] Netflix, Feign makes writing java http clients easier, https://github.com/OpenFeign/feign, 2 Mayıs 2017.

ÖZGEÇMİŞ

KİŞİSEL BİLGİLER

Adı Soyadı : Tuğrul AŞIK

Doğum Tarihi ve Yeri : 01.01.1989, Balıkesir

Yabancı Dili : İngilizce, Almanca

E-posta : tugrulasik@gmail.com

ÖĞRENİM DURUMU

Derece	Alan	Okul/Üniversite	Mezuniyet Yılı
Lisans	Bilgisayar Mühendisliği Bölümü	Ege Üniversitesi	2012
Lise	Fen Bilimleri	Sırrı Yırcalı Anadolu Lisesi	2007

İŞ TECRÜBESİ

Yıl	Firma/Kurum	Görevi
2017 - ...	Doğuş Planet A.Ş (n11.com)	Kıdemli Yazılım Mühendisi
2013-2017	Merkezi Kayıt İstanbul A.Ş	Uzman Yazılım Mühendisi
2012-2013	Avea	Teknik Ürün Geliştirme Uzmanı

YAYINLARI

Bildiri

Asik T. ve Selcuk Y.E., (2017). “Policy Enforcement upon Software Based on Microservice Architecture”, Software Engineering Research, Management and Applications (SERA), 2017 IEEE 15th International Conference, 7-9 June 2017, London, UK.