

T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

MİKROSERVİSLERİN KEŞİFSEL SINANMASI VE
GÖRSELLEŞTİRİLMESİ

Mustafa Bedri EĞRİLMEZ

YÜKSEK LİSANS TEZİ

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

Danışman

Dr. Öğr. Üyesi Yunus Emre SELÇUK

Temmuz, 2020

T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

MİKROSERVİSLERİN KEŞİFSEL SINANMASI VE
GÖRSELLEŞTİRİLMESİ

Mustafa Bedri EĞRİLMEZ tarafından hazırlanan tez çalışması 16.7.2020 tarihinde aşağıdaki jüri tarafından Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı, Bilgisayar Mühendisliği Programı **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Dr. Öğr. Üyesi Yunus Emre SELÇUK

Yıldız Teknik Üniversitesi

Danışman

Jüri Üyeleri

Dr. Öğr. Üyesi Yunus Emre SELÇUK, Danışman

Yıldız Teknik Üniversitesi

Prof. Dr. Oya KALIPSIZ, Üye

Yıldız Teknik Üniversitesi

Prof. Dr. Selim AKYOKUŞ, Üye

İstanbul Medipol Üniversitesi

Danışmanım Dr. Öğr. Üyesi Yunus Emre SELÇUK sorumluluğunda tarafımda hazırlanan Mikroservislerin Keşifsel Sınanması ve Görselleştirilmesi başlıklı çalışmada veri toplama ve veri kullanımında gerekli yasal izinleri aldığımı, diğer kaynaklardan aldığım bilgileri ana metin ve referanslarda eksiksiz gösterdiğimi, araştırma verilerine ve sonuçlarına ilişkin çarpıtma ve/veya sahtecilik yapmadığımı, çalışmam süresince bilimsel araştırma ve etik ilkelerine uygun davrandığımı beyan ederim. Beyanımın aksinin ispatı halinde her türlü yasal sonucu kabul ederim.

Mustafa Bedri EĞRİLMEZ

İmza



*Sevgili eşim
ve
kızlarımıza*

TEŞEKKÜR

Tez çalışması sürecinde bana her konuda destek olan ve yol gösteren, beni her zaman çalışmaya teşvik eden, tecrübe ve bilgisini esirgemeyen danışmanım Sayın Dr. Öğr. Üyesi Yunus Emre SELÇUK'a teşekkür ve minnetlerimi sunarım. Onun değerli katkıları olmasaydı böyle bir çalışma ortaya çıkamazdı.

Hayatta ihtiyaç duyduğum her anda beni koşulsuz destekleyen ve moral veren, herşeyden çok sevdiğim eşim ve kızlarımıza sonsuz sevgilerimi sunarım.

Mustafa Bedri EĞRİLMEZ



İÇİNDEKİLER

KISALTMA LİSTESİ	ix
ŞEKİL LİSTESİ	x
TABLO LİSTESİ	xii
ÖZET	xiii
ABSTRACT	xv
1 GİRİŞ	1
1.1 Literatür Özeti.....	2
1.2 Tezin Amacı	6
1.3 Hipotez	6
1.4 Tezin İçeriği	6
2 MİKROSERVİSLER	8
2.1 Mikroservislerin Gelişiminde Rol Oynayan Teknolojiler	9
2.2 Mikroservislerin Çalışma Mantığı	11
3 DAĞITIK İZLEYİCİLER	13
3.1 Günlük Yapısı ve Enstrümantasyon	14
3.2 Dağıtık İzleyici Terminolojisi.....	15
3.3 Olay Bilgilerinin Toplanması.....	16
3.4 Dağıtık İzleyicilerde Enstrümantasyon	17
3.5 Dağıtık İzleyicilerin Çalışma Prensibi	18
3.6 Dağıtık İzleyicilerin Kullanıcı Arayüzleri	20
3.7 Örnekleyiciler.....	22
3.8 Yayılım.....	22
3.9 Standartlaştırma Çalışmaları.....	24

4 YAZILIM GÖRSELLEŞTİRME	26
4.1 Sıralama Diyagramları	27
4.2 Alternatif İz Görselleştirme Yöntemleri	30
4.2.1 Akış Grafiği.....	30
4.2.2 Odak Grafiği.....	31
4.2.3 Çağrı Bağlam Ağacı	31
4.2.4 Gantt Şeması	31
4.2.5 Petri Ağı	32
5 KEŞİF SINAMASI	33
5.1 Keşif Sınavının Özellikleri.....	35
5.2 Keşif Sınavının Temel İhtiyaçları	36
6 XPLORA UYGULAMASI MİMARİSİ	37
6.1 XploraServiceLib Bileşeni	38
6.2 SequenceDiagramLib Bileşeni	39
6.2.1 Sıralama (Sequence) Sınıfı	39
6.2.2 Katılımcı (Participant) Sınıfı	40
6.2.3 İleti (Message) Sınıfı.....	42
6.2.4 Etkinleşme Kutusu (Activation Box) Sınıfı	44
6.2.5 Kutu (Box) Sınıfı	45
7 XPLORA ARACI KULLANIM SENARYOLARI	47
7.1 Senaryo 1 – Bir İstemci ve Mikroservis Arasında Etkileşim	48
7.2 Senaryo 2 – Dinleyici Enstrümantasyonu ile İz Oluşturma	51
7.3 Senaryo 3 – İki Servis ve İstemci Arasındaki Etkileşim	54
7.4 Senaryo 4 – Bağlantı Problemleri ve Devre Kesiciler	57

8 BENZER ÇALIŞMALAR İLE KARŞILAŞTIRMA	62
8.1 XPLORA ve Dağıtık İzleyiciler	62
8.2 XPLORA ve Keşifsel Sınama Araçları	63
8.3 XPLORA ve Sıralama Diyagramı Kullanan Görselleştirme Araçları	65
9 SONUÇ VE ÖNERİLER	68
9.1 Sıralama Diyagramlarının Daha Etkin Kullanımı	70
9.2 Sıralama Diyagramına Alternatif Diyagram Tipleri.....	70
9.3 Ek Senaryolar.....	71
9.4 Büyük İzler için Anlaşılabilir Görselleştirme	71
9.5 Aracın Sağladığı Faydanın Nicel Ölçümü	71
KAYNAKÇA	72
A UBER JAEGER DAĞITIK İZLEYİCİ UYGULAMASI	78
A.1 Kurulum.....	78
A.2 Arayüz.....	79
B SÖZLÜK	81
TEZDEN ÜRETİLMİŞ YAYINLAR	87

AOP	Aspect Oriented Programming
CASE	Computer-Aided Software Engineering
FAAS	Function as a Service
HTTP	Hyper Text Transfer Protocol
ISTQB	International Software Testing Qualifications Board
JSON	JavaScript Object Notation
JVMPI	Java Virtual Machine Profiler Interface
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
TWAIN	Technology without an Interesting Name
UML	Unified Modeling Language
WCF	Windows Communication Foundation
WOSA	Windows Open Systems Architecture

ŞEKİL LİSTESİ

Şekil 1.1 Tezin dayandığı yazılım disiplinleri.....	2
Şekil 2.1 Mikroservislerin yekpare mimariyle karşılaştırılması.....	8
Şekil 2.2 Mikroservislerin gelişiminde rol oynayan teknolojiler	10
Şekil 2.3 Mikroservis mimarisine göre tasarlanmış bir uygulama.....	11
Şekil 3.1 Sıradan bir günlük örneği	13
Şekil 3.2 Günlükler için örnek enstrümantasyon kodu ve çıktısı	14
Şekil 3.3 Dağıtık izleyicilerde kullanılan kavramlar	15
Şekil 3.4 Dağıtık izleyiciler için örnek enstrümantasyon kodu	17
Şekil 3.5 Twitter Zipkin dağıtık izleyicisine ait diyagram.....	18
Şekil 3.6 Twitter Zipkin dağıtık izleyici.....	20
Şekil 3.7 Uber Jaeger dağıtık izleyici	21
Şekil 3.8 Dağıtık izleyici – Kibana entegrasyonu	21
Şekil 3.9 Bilgisayar ağı yayılımı	23
Şekil 3.10 İşlem içi yayılım	23
Şekil 3.11 OpenTracing destekli Jaeger dağıtık izleyicisine ait diyagram.....	25
Şekil 4.1 Kodun şehir, molekül ve takımda metaforlarıyla görselleştirilmesi...	27
Şekil 4.2 Sıralama diyagramlarının UML diyagramları arasındaki yeri	27
Şekil 4.3 Örnek bir sıralama diyagramı.....	29
Şekil 4.4 Alternatif yürütüm görselleştirme yöntemleri.....	30
Şekil 5.1 Keşif sınavasının betikleştirilmiş sınamayla karşılaştırılması	33
Şekil 5.2 Keşif sınavasının ISTQB eğitim müfredatındaki yeri.....	34
Şekil 5.3 Betikleştirilmiş sınama - Keşif sınavası sürekliliği.....	35
Şekil 6.1 XPLORA uygulaması bileşen diyagramı	37

Şekil 6.2 XPLORA uygulaması sınıf diyagramı	38
Şekil 6.3 Katılımcı sınıfının görselleştirilmesi.....	40
Şekil 6.4 İleti sınıfının görselleştirilmesi	42
Şekil 6.5 Etkinleşme kutusu sınıfının görselleştirilmesi.....	44
Şekil 6.6 Kutu sınıfının görselleştirilmesi	45
Şekil 7.1 Enstrümante edilmiş bir uygulama ve XPLORA aracına ait diyagram .	47
Şekil 7.2 İstemci-mikroservis senaryosuna ait bileşen diyagramı	48
Şekil 7.3 İstemci-mikroservis senaryosunun XPLORA görselleştirmesi	49
Şekil 7.4 İstemci tarafında yapılan enstrümantasyon	50
Şekil 7.5 Sunucu tarafında yapılan enstrümantasyon	50
Şekil 7.6 Dinleyici enstrümantasyonu ile iz oluşturma.....	51
Şekil 7.7 Dinleyici enstrümantasyonu senaryosu bileşen diyagramı.....	52
Şekil 7.8 Dinleyici enstrümantasyonu senaryosu XPLORA görselleştirmesi	53
Şekil 7.9 İz etiketlerinin görselleştirilmesi	54
Şekil 7.10 Üç bileşen senaryosuna ait bileşen diyagramı.....	55
Şekil 7.11 Üç bileşen senaryosuna ait son kullanıcı ekranı.....	56
Şekil 7.12 Üç bileşen senaryosunun XPLORA görselleştirmesi	57
Şekil 7.13 Devre kesici.....	58
Şekil 7.14 Devre kesici senaryosuna ait bileşen diyagramı	59
Şekil 7.15 Devre kesici senaryosuna ait önyüz ekranı	59
Şekil 7.16 Devre kesici senaryosunun XPLORA görselleştirmesi.....	61
Şekil 8.1 Çalışmanın standart bir dağıtık tarayıcıyla karşılaştırılması.....	63
Şekil A.1 Uber Jaeger kurulum paketi içeriği	78
Şekil A.2 Uber Jaeger ana sorgulama ekranı.....	79
Şekil A.3 Uber Jaeger iz detay ekranı	80

TABLO LİSTESİ

Tablo 3.1 Dağıtık izleyiciler ve kullanım amaçları	19
Tablo 4.1 Sıralama diyagramı öğeleri	28
Tablo 8.1 Keşifsel sınaama araçları.....	64
Tablo 8.2 Sıralama diyagramı kullanan görselleştirme araçları.....	66
Tablo 8.3 Benzer görselleştirme çalışmaları ile karşılaştırma	67
Tablo 9.1 Çalışmanın ilgili yazılım mühendisliği alanlarına sağladığı katkı	69



Mikroservislerin Keşifsel Sınanması ve Görselleştirilmesi

Mustafa Bedri EĞRİLMEZ

Bilgisayar Mühendisliği Anabilim Dalı

Yüksek Lisans Tezi

Danışman: Dr. Öğr. Üyesi Yunus Emre SELÇUK

Mikroservisler yakın dönemde popülerlik kazanmış, alışlagelmiş yekpare mimarilerden farklı olarak bir yazılım uygulamasını yapıtaşlarına ayırmayı prensip edinmiş bir yazılım mimari stilidir. Uygulamayı oluşturan her bir bileşenin karmaşıklığının en alt seviyeye indirilmesi, ölçeklenebilirliğinin kolaylığı, devreye almaya getirdiği esneklikler gibi daha ilk bakışta göze çarpan birçok artıyı bünyesinde bulundurur. Bununla birlikte, her derde deva bir çözüm de değildir: Birçok ayrı modüle, sisteme ve konuma parçalanmış bir uygulamanın takibi ve sınanması zorlukları beraberinde getirir.

Yazılım günlükleri, yazılım geliştirme ve bakım sürecinde hata ayıklama, güvenlik ve uygunluk için tercih edilen geleneksel yöntem olmuşlardır. Fakat günlüklerin düz yapısı ve uygulama bileşenleri arasında bağlam bilgisi paylaşım mekanizmaları bulundurmamaları, günlüklerin mikroservisler gibi modern çok işlemcili, dağıtık mimarilerde kullanımının önünde bir engel oluşturmaktadır.

Yazılım mühendislerinin bu engellere karşı çözümü, düz günlük oluşturma hizmetlerini hiyerarşik yapıya dönüştürmek ve bileşenler arasında yürütüm bağlam bilgisinin nakil edilebilmesi gibi ek becerilerle zenginleştirmek olmuştur. Bu ek işlevleri sağlayan uygulamalara dağıtık izleyiciler adı verilmiştir ve modern

büyük ölçekli yazılım sistemlerinin olmazsa olmazlarından biri haline gelmişlerdir. Dağıtık izleyicilerin çalışma mantığı, inceleme altındaki uygulamanın tüm bileşenleri tarafından oluşturulan günlük olaylarının daha sonra ihtiyaç duyulduğunda analiz edilebilmesi ve görselleştirilebilmesi için yakalanması ve bir merkezde saklanmasına dayanır.

Bununla birlikte, dağıtık izleyicilerce oluşturulmuş mekanizmalar alternatif bir şekilde de kullanılabilirler: Bu çalışmada tanıtılan XPLORA aracı, mikroservis mimarisine dayanan uygulamaların yürütümünü bir sıralama diyagramı üzerinde görselleştiren bir dinamik sınaama aracıdır. Araç, açık ve sağlayıcı bağımsız bir dağıtık izleyici standardı olan OpenTracing çerçevesine dayanmaktadır. Bu da aracın OpenTracing’le uyumlu hale getirilmiş herhangi bir uygulama ile kolaylıkla entegre edilebilmesini mümkün kılar. XPLORA’nın geleneksel izleyicilere getirdiği yenilik, yakalanan iz olaylarını gerçek zamanda görselleştirilen duraksatma noktaları olarak kullanabilmesidir.

Anahtar kelimeler: Mikroservisler, dağıtık tarayıcılar, dinamik program görselleştirme, keşif sınaması

Exploratory Testing and Visualization of Microservices

Mustafa Bedri EĞRİLMEZ

Department of Computer Engineering

Master of Science Thesis

Advisor: Asst. Prof. Yunus Emre SELÇUK

Microservices is an emerging software architectural style which is distinguished from its monolithic counterpart by decomposing a software application into its many building blocks. Gains such as reduction of individual module complexity, ease of scalability, and deployment flexibility are obvious. However, it is not a magic bullet: Testing and monitoring a system that has fragmented into multitudes of disparate modules, systems, and locations brings new challenges.

Application logging has been the traditionally preferred method for debugging, security, and compliance for software development and maintenance process. But the flat nature of logging and the absence of mechanisms for sharing context information between the solution's components stay as obstacles against their usage in modern multiprocessing, distributed architectures.

Software engineers' answer to these challenges has been to enhance established logging facilities with additional capabilities such as moving from flat to hierarchical log records organization and adding mechanisms for transferring context information within components. The family of applications that implement this functionality is called distributed tracers, and have become a necessity for modern huge scale software systems. The typical operation of the

distributed tracers involves capturing and storing log events generated by all components of the solution in a central location, so they can be analyzed and visualized as needed retrospectively.

However, the mechanisms set out by distributed tracers can be used in an alternative way: XPLORA tool, which is presented in this study, is a dynamic testing tool visualizing the execution of a microservices solution in a sequence diagram. The tool builds upon OpenTracing, an open and vendor neutral distributed tracing framework. The usage of the framework allows the tool to be integrated into any OpenTracing compatible application with ease. XPLORA enhances traditional tracers by making use of the captured events as breakpoints that are visualized in real time.

Keywords: Microservices, distributed tracers, dynamic program visualization, exploratory testing

1 GİRİŞ

Yazılım mimarilerinin gelişiminde yekpare mimariden hizmet odaklı mimariye, son olarak da mikroservislere uzun bir yol katedilmiştir. Günümüzde gelinen noktada uygulamaların iyi tanımlanmış tek bir fonksiyonu yerine getirmekten sorumlu, birbiriyle gevşek bağlarla bağlı, her biri kendi verisini kontrol eden çok adette bileşen olarak oluşturulması yönünde belirgin bir eğilim gözlemlenmektedir. Bu eğilimin ulaştığı doruk noktası mikroservis yazılım mimarisidir.

Mikroservis mimarisi çözümü oluşturan bağımsız bileşenlerin her birinin karmaşıklığını en aza indirmeyi başarırken, diğer taraftan az sayıda hatta tek bir bileşenden oluşmuş yazılım uygulamasının, sayısı binlere ulaşabilen adetlerde bileşene bölünecek şekilde parçalanmasına sebep olur. Bu da, yürütülen her bir metodun çalışmasının farklı sunuculardaki farklı hizmetlere ve işlemlere bölünmesine ve uygulamaların sına ve izlenmesini zorlaştırır.

Yazılım mühendislerinin bu soruna getirdiği çözüm, programlamanın ilk zamanlarından beri varolan günlük oluşturma mekanizmasının düz veri yapılarını hiyerarşik yapıya çevirmek ve yürütüm bağlam bilgisinin bağımsız bir bileşenden diğerine naklini sağlayan teknikler gibi ek becerilerle zenginleştirmek olmuştur. Bu ek fonksiyonalityi sağlayan uygulamalara dağıtık izleyiciler adı verilmiştir, varoluş sebepleri de büyük ölçekli yazılım sistemlerinin duyduğu gereksinimdir.

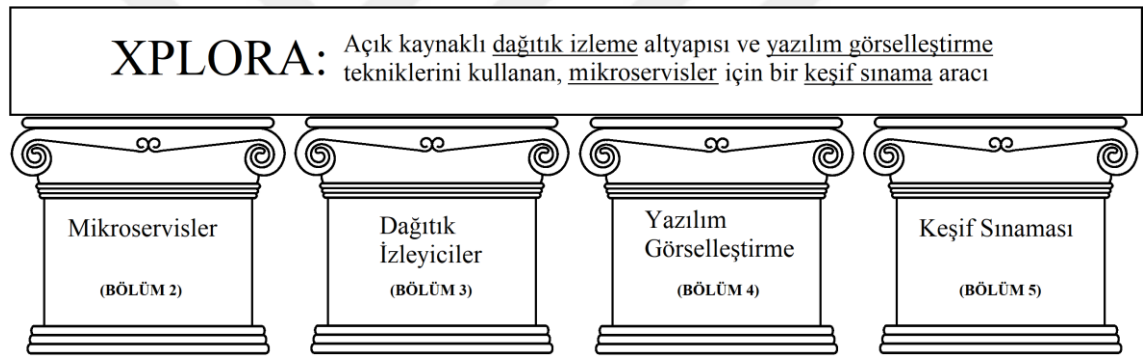
Dağıtık izleyicilerin alışlagelmiş çalışma şekli, inceleme altındaki uygulama tarafından üretilen günlük olaylarının yakalanması ve sonradan ihtiyaca göre analiz edilebilmesi ve görselleştirilebilmesine olanak sağlayacak şekilde depolanmasından ibarettir.

Buna karşın, dağıtık izleyicilerce belirlenmiş mekanizmalar alternatif bir şekilde de kullanılabilir. Bu çalışmada, sözü edilen aynı günlük olaylarının hazırladığımız

keşif sınaması aracına yönlendirilmesi yoluyla inceleme altındaki uygulamanın izlenmesini, analiz edilmesini ve üzerinde değişiklik yapılmasını mümkün kılan bu alternatif çözüm sunulacaktır. Araç, açık ve sağlayıcı bağımsız bir dağıtık izleyici standardı olan OpenTracing'e dayanmakta ve uygulamanın yürütümünü gerçek zamanlı görselleştirebilmesi ve denetimini sağlamaktadır.

1.1 Literatür Özeti

Tez çalışması, yazılım mühendisliğinin dört farklı disiplinini bir araya getirmektedir. Her bir disipline ait literatür özeti, kendi bölümünde ayrı ayrı ele alınmıştır.



Şekil 1.1 Tezin dayandığı yazılım disiplinleri

Mikroservisler: Mikroservisler, yazılım uygulamasını çok sayıda otonom bileşene bölerek uygulama karmaşıklığını azaltmayı amaçlayan bir yazılım mimari şeklidir (Richardson, 2019).

Martin Fowler'a (2014) göre mikroservis teriminin ilk kullanılması Mayıs 2011'de bir yazılım mimari çalıştayında gerçekleşmiştir. Bu tarihin öncesine bakıldığında, daha 2006 (Gray) yılında Amazon firmasından Werner Vogels'ın "Bizim için hizmet tabanlığının anlamı: veriyi, erişimin sadece yayınlanmış bir hizmet arayüz üzerinden olacak şekilde, veri üzerinde işlem yapan iş mantığıyla sarmalamak demektir. Hizmetin dışından doğrudan veritabanı erişimine izin verilmez ve

hizmetler arasında veri paylaşımı yoktur” ve Netflix firmasından Adrian Cockfort’un “sınırlandırılmış bağlamli gevşek bağli hizmet tabanlı mimari” ifadeleri mikroservislerin öncü tanımlarıdır.

Mikroservis mimarisinin geçen zaman içinde benimsenmesi ve olgunlaşmasıyla kendisine özgü tasarım örüntüleri ortaya çıkmıştır (Richardson, 2019).

Dağıtık izleyiciler: Mikroservisler gibi modern dağıtık mimariye göre tasarlanmış uygulamaların analiz edilebilmelerini ve gözlemlenebilmelerini amaçlayan yardımcı uygulamalardır. Sınama altındaki uygulama tarafından üretilmiş günlük olaylarının toplanma, depolanma, sorgulanma ve görselleştirme fonksiyonlarını yerine getirirler.

Yeni bir araştırma alanı olmasından dolayı, sırf dağıtık izleyicileri konu edinmiş sadece iki kitap bulunmaktadır (Shkuro, 2019) (Parker, Spoonhower, Mace & Isaacs, 2020). Dağıtık izleyicileri en kapsamlı şekilde inceleyen makale ise, Carnegie Mellon Üniversitesi Paralel Veri Laboratuvarına bağli araştırmacıların yayınladığı 25 sayfalık makaledir (Sambasivan, Fonseca, Shafer & Ganger, 2014).

End-to-End’in kısaltmasından gelen ETE (Hellerstein, Maccabee, Mills & Turek, 1999) uçtan uca iz ağacı oluşturabilen dağıtık izleyici uygulamalarının ilklerindır. Uygulamanın diğer izleyicilerden ayrılan özelliğı, yakaladığı ham olayları “işlem aracı” adlı modülüne yönlendirmesidir. Modülün özelliğı kendi kural dilinin bulunmaktadır. Kullanıcı bu dili kullanarak, olay-kapsam bağlantısını inceleme altındaki uygulama üzerinde değışiklik yapmadan bağımsız bir şekilde tanımlayabilir.

Pinpoint (Chen, Kiciman, Fratkin, Fox & Brewer, 2002), sistemin doğru çalışıp çalışmadığını ve hata kaynağını bulmaya yoğunlaşmış bir dağıtık izleyici uygulamasıdır.

Magpie (Barham, Isaacs, Mortier & Narayanan, 2003), kod üzerinde değışiklik yapmadan dağıtık sistem performans ölçümü üzerine yoğunlaşmış bir dağıtık izleyici uygulamasıdır. Sistemin doğru çalışıp çalışmadığının yanı sıra, performans problemleriyle de ilgilenir.

Stardust'ın (Thereska vd., 2006) diğer dağıtık izleyicilerden daha özelleşmiş bir amacı bulunmaktadır: Dağıtık bir uygulama tarafından yürütülen herhangi bir metodun yürütümüne, sistemin birçok donanım ögesi dahil olur. Stardust'un hedefi işletim sırasında kullanılan kaynakları kullanan varlıklarla eşleştirmektir.

X-Trace (Fonseca, Porter, Katz, Shenker & Stoica, 2007) Berkeley Üniversitesinde geliştirilmiş bir dağıtık izleyici uygulamasıdır. HTTP tabanlı dağıtık uygulamaların yayılımı için üstbilgilerin kullanımı ilk kez bu çalışmada gerçekleştirilmiştir.

Bir başka dağıtık izleyici uygulaması olan Google Dapper (Sigelman, vd. 2010) uygulaması, Google'ın büyük ölçekli sistemlerinde başarıyla kullanılmış ve modern dağıtık izleyicilerin kullandığı terminoloji için belirleyici olmuştur.

Dağıtık izleyicilerin her birinin farklı enstrümantasyon koduna ihtiyaç duyacak şekilde tasarlanmış olması, yaygınlaşmalarının önünde bir engeldir. Bu problemin önüne geçmek için OpenTracing (2020) ve OpenCensus (2020) adlı açık kaynaklı dağıtık izleyici standartları geliştirilmiştir. Uber Jaeger (2020), Uber tarafından geliştirilmiş bir dağıtık izleyici uygulamasıdır. Önemi, standart destekli ilk dağıtık izleyici uygulaması olmasından gelir.

Yazılım Görselleştirme: Bir uygulamanın yürütümünü görselleştirerek anlaşılabilir hale getirmek, yazılım mühendislerinin önemli bir hedefidir. Yürütüm görselleştirme teknikleri, statik ve dinamik olarak ikiye ayrılır. Dinamik analizin gücü uygulamanın gerçek çalışmasını modelleyebilmesi, zayıflığı ise sadece yürütülmüş senaryoları kapsayabileceğinden dolayı kaplam alanının kısıtlı kalmasıdır.

Bir uygulamanın yürütümünü molekül yapısından bir şehrin mahallelerine kadar farklı şekillerde modelleyerek görselleştirmeyi amaçlamış çok yaratıcı çalışmalar bulunmaktadır. Bu konudaki literatür araştırmamız, uygulama yürütümünü sıralama diyagramı olarak modelleyen çalışmalarla sınırlı tutulmuştur.

SPIDOR uygulaması (Malloy & Power, 2005), C++ dilinin boyuta yönelik programlama özelliklerini kullanarak yakalayarak sıralama diyagramları aracılığı ile görselleştiren bir uygulamadır.

Form çerçevesi (Souder, Mancoridis & Salah, 2001), bir Java kütüphanesi olan JVMPI aracılığı ile, Java uygulamalarının sıralama diyagramları şeklinde görselleştirilmesini amaçlamaktadır. Form çerçevesi, karşılaştıklarımızın arasında bu tez çalışmasına çıktı yönünden en benzer çalışmadır. Ancak bir mikroservis uygulamasının ihtiyaçlarını karşılayamamaktadır.

SDR çerçevesi (Cornelissen, van Deursen, Moonen & Zaidman, 2007) de uygulamanın dinamik yürütümünü sıralama diyagramları ile görselleştirilmesini amaçlayan bir diğer çalışmadır. İncelencek yürütümün büyük ölçekli olması durumunda da kullanışlı ve anlaşılabilir bir görselleştirme sağlanabilmesi için gerekli soyutlama ve filtreleme özellikleri üzerinde durur.

Keşif sınaması: Sınayıcıların kendi bilgi, tecrübe ve sezgilerine dayanarak sına altındaki sistemdeki hataları tespit edebilmek amacıyla sistemle etkileşim içinde bulundukları bir sına çeşididir. Temel prensipleri “eş zamanlı öğrenme, sına tasarımı ve test yürütümü” olarak ifade edilebilir (Bach, 2002). Bir özelliğin amaçlandığı şekilde çalışıp çalışmadığının tespiti için insan muhakemesini kullanır. Doğaçlama doğasıyla yazılım endüstrisinde kendisine geniş bir kullanıcı kitlesi bulmuştur.

Keşif sınamasının terim olarak çıkışı 1984 yılında (Kaner, 2008), yazılı olarak ilk kullanımı ise Cem Kaner’in (1997) “Bilgisayar Yazılımın Sınanması” adlı kitabıyla olmuştur. Kitap betikleştirilmiş sınamalar tamamlandıktan sonra sına sürecinin devam etmesi önerirken, henüz istikrarlı bir duruma gelmemiş bir bilgisayar yazılımının sına tasarım ve dokümantasyon geçerliliğini kısa bir süre içinde kaybedeceğinden, bu ikisine çok fazla zaman ayrılmasına karşı uyarır.

Craig ve Jaskiel (2002), keşif sınamasının sınavıcının çalışmasını hemen fayda görebileceği kısımlar üzerinde yoğunlaştırabilmesine imkan sağladığının altını çizer. Betikleştirilmiş sınamanın doğası gereği, yapılan birçok sınamanın hata ortaya çıkarmadığını, bu nedenle de herhangi bir amaca hizmet etmediğinin altını çizer.

1.2 Tezin Amacı

Tezin amacı, mikroservis gibi dağıtık mimarideki yazılım uygulamalarının yürütümünün görselleştirmesi için yeni bir yöntem sağlayarak, yazılım sınama sürecine katkıda bulunmaktır.

Tezin hedefi, bu görselleştirmeyi farklı kullanım senaryoları için, sıralama diyagramı gibi yaygın olarak bilinen bir diyagram tipi kullanan somut bir uygulamayla sunmaktır.

1.3 Hipotez

Günlük kayıtları, dağıtık izleyicilerin sınama altındaki uygulamayı daha sonradan analiz edebilmeleri için üretilen kayıtlardır. Tezin ana hipotezi, sonradan kullanım için üretilen bu günlük kayıtlarının, hazırlanacak özel amaçlı bir keşif sınaması aracı ile uygulama yürütümünün dinamik olarak görselleştirilmesi ve denetimi için de kullanılabileceğidir. Böylece sınama sürecine katkıda bulunmaları mümkün olacaktır.

Tezin alt hipotezi, enstrümantasyonu mümkün olmayan çözüm bileşenlerinin de, kullandıkları paylaşımlı kütüphanelerin enstrümantasyonu yapılarak iz grafiğine dahil edilebilecekleridir. Bölüm 7.2'deki senaryo bu hipotez üzerine kurulmuştur.

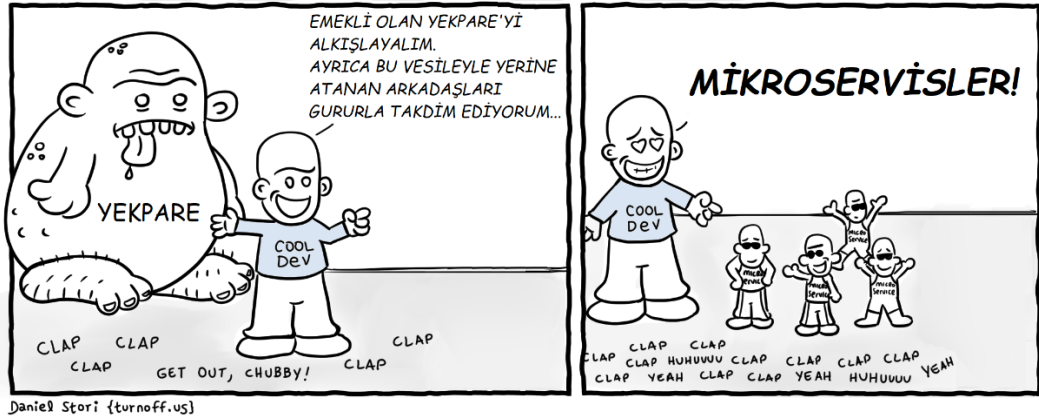
1.4 Tezin İçeriği

Tez metni şu şekilde bölümlenmiştir: Giriş bölümünde tezin dayandığı dört yazılım disiplini olan mikroservisler, dağıtık izleyiciler, dinamik yazılım yürütüm görselleştirme ve keşif sınaması konuları üzerinde genel bir literatür araştırması yapılmıştır. İkinci bölümde mikroservislerin yapısı üzerinde çalışma ve araştırmaya yer verilmiştir. Üçüncü bölümde dağıtık izleyicilerin çalışma prensipleri ele alınmış ve yaygın olarak kullanılmakta olan dağıtık izleyiciler tanıtılmıştır. Dördüncü bölümde dinamik yazılım yürütüm görselleştirmesi için kullanılan sıralama diyagramları ve alternatif grafik tipleri özetlenmiştir. Beşinci

bölümde keşif sınavının kullanım alanları ve gelişimi anlatılmıştır. Altıncı bölümde XPORA uygulamasının genel mimari yapısına değinilmiştir. Yedinci bölümde teze konu olan XPORA uygulamasının çalışması dört farklı senaryoyla tanıtılmıştır. Sekizinci bölümde tez çalışmasına dair gelecek için planlanan yol için önerilerde bulunulmuştur. Son bölümde ise sonuç ve araştırma sorularına verilen yanıtlara yer verilmiştir.

Ek bölümlerin ilkinde, yaygın olarak kullanılan bir dağıtık izleyici olan Uber Jaeger'in (2020) kurulumu, kullanımı ve özellikleri özetlenmiş, XPORA aracıyla karşılaştırılması yapılmıştır. Çalışmada kullanılan birçok terimin kabul görmüş Türkçe karşılığına rastlanılamamıştır. İkinci ek bölümde bu tip terimler ve tercih edilmiş Türkçe karşılıklarına yer verildiği bir sözlük oluşturulmuştur.

Mikroservis mimarisi, yazılım uygulamasının iş dağılımı içinde her birinin tek bir görevi yerine getirdiği otonom bileşenlere ayrıştırıldığı yazılım mimari stilidir. Yekpare bir çözümden farklı olarak, bu bileşenler olabildiğince gevşek-bağlı olarak tasarlanır. İki mimarinin yaklaşımlarındaki fark, bir karikatüristin gözünden Şekil 2.1'deki gibi resmedilmiştir (Stori, 2017). Ayrım veri kalıcılığı katmanına da yansır: Ortak veritabanı, yerini her mikroservisin kendi verisini yönettiği ve erişebildiği bir veritabanı organizasyon yapısına bırakır.



Şekil 2.1 Mikroservislerin yekpare mimariyle karşılaştırılması

Bu bağımsızlık, yazılım geliştiricilerin her mikroservis için en uygun yazılım dili ve geliştirme ortamını seçebilmesine, proje yöneticilerinin iş bölümünü en uygun şekilde yapabilmelerine olanak sağlar. Hizmet yöneticileri için ise cazip yönü bileşenlerin birbirlerini etkilemeden, birbirinden bağımsız olarak kurulabilmeleri ve güncellenebilmeleridir.

Programlama dilinden bağımsız, sade yapıdaki RESTsel HTTP çağrıları, mikroservisler arasında tercih edilen haberleşme yöntemi olarak öne çıkmıştır. Bu

iletiřim tipi, karmařık ve pahalı kurumsal veri yolu kullanan servis odaklı mimariye gre avantajlar ierir.

Olumsuz ynleri incelendiėinde, mikroservis mimarisinin daėıtık doėası (aė arızaları ve gecikme sreleri gibi) bilgisayar aėı ve (tek bir aėrının ok adette ayrı sistem zerinden geeceėi iin) sına ve hata ayıklama ile ilgili konuların, nceki yazılım mimarilerine dayanan uygulamalara gre daha dikkatli tasarlanmalarını gerekli kılar.¹

Mikroservislerin yaygınlařmasında aynı dnemde ortaya ıkan diėer birok yazılımsal geliřme anahtar rol oynamıřtır: Sınırlandırılmıř yapısal baėlımlar, kesintisiz yazılım tmleřtirmesi ve alan odaklı tasarım bu geliřmelerin nde gelenleridir. Mikroservisleri yakından etkilemiř diėer geliřmeler bařarısızlık iin tasarım, veri yalıtımı, altyapı otomasyonu, leklendirilmıř eviklik, apraz fonksiyonlu takımlar ve bařtan sonra rn sahipliėidir. Bu yaklařımlar, daėıtık aė ile ilgili bir ok problemi zmenin yanında, byk lekli firmaların karřılařtıėı organizasyonel problemlerin stesinden gelinmesinde de bařarılı olmuřtur.

2.1 Mikroservislerin Geliřiminde Rol Oynayan Teknolojiler

Mikroservislerin geliřiminde řekil 2.2'deki on teknoloji kilit rol oynamıřtır (Jamshidi, Pahl, Mendonca, Lewis & Tilkov, 2018, s. 26). Bunların beřinin ortaya ıkıřı, mikroservis teriminin kullanımından nceye denk gelir:

Barındırıcılařtırma: Bileřenlerin zerlerinde alıřtıkları iřletim sisteminden soyutlar ve birbirinden baėımsız olarak paketlenmesine, kurulmasına ve ynetilmesine olanak saėlar.

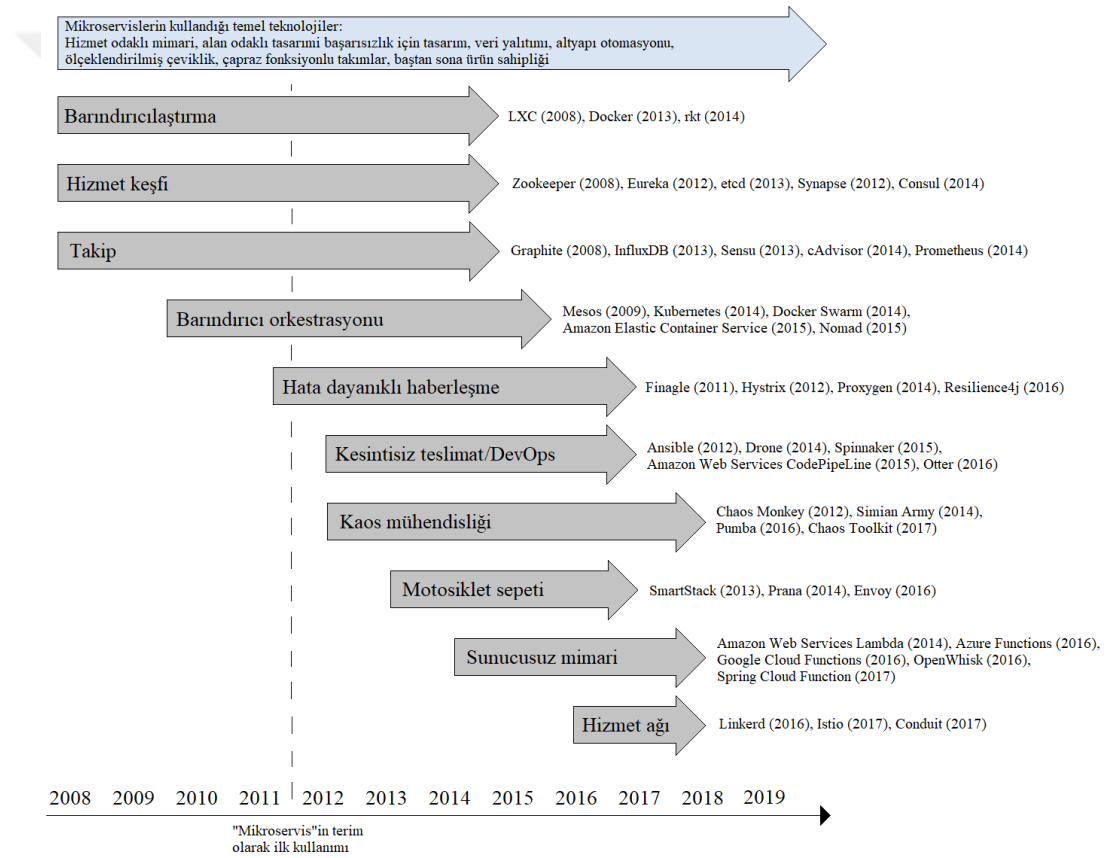
Hizmet keřfi: Bileřenlerin, birbirlerinin bilgisayar aėındaki tam konumlarını bilmeden haberleřebilmelerini amalar.

¹ “Daėıtık sistem: Varlıėından haberinizin bile olmadıėı bir bilgisayarın kendi bilgisayarınızı kullanılamaz hale getirebildiėi sistem” Leslie Lamport, bilim insanı, daėıtık sistemler uzmanı ve LaTeX sisteminin mucidi

Takip: Bileşenlerin davranışının canlı ortamda gözlenmesine ve analizine olanak sağlar.

Barındırıcı orkestrasyonu: Barındırıcılar, uygulamayı bulundukları bilgisayar ortamından soyutlayan yapılardır. Barındırıcı senkronizasyonu, barındırıcıların tahsisi ve yönetimiyle ilgili işlerin otomasyonuna olanak sağlar.

Hata dayanıklı haberleşme: Hizmetlerin daha verimli ve güvenilir bir şekilde haberleşmesini sağlayan, ağ gecikme ve hata durumlarına karşı dayanıklı haberleşme kütüphanelerini içerir.



Şekil 2.2 Mikroservislerin gelişiminde rol oynayan teknolojiler

Diğer beş teknolojinin ortaya çıkması ise mikroservis kavramının kullanılmaya başlamasından sonraya denk gelir. Temel amaçları mikroservislerin kullanımını kolaylaştırmaktır:

Kesintisiz teslimat/DevOps: Yazılım geliştiriciler ve operasyon personeli arasındaki boşluğu kapatmayı sağlayan bir yazılım geliştirme stratejisidir. Mikroservislerin kolaylıkla geliştirilip devreye alınmasını amaçlar.

Kaos mühendisliği: Hata ve saldırı içitmesi gibi kritik güvenilirlik ve güvenlik sınaama tekniklerinin yürütümünün otomasyonunu amaçlar.

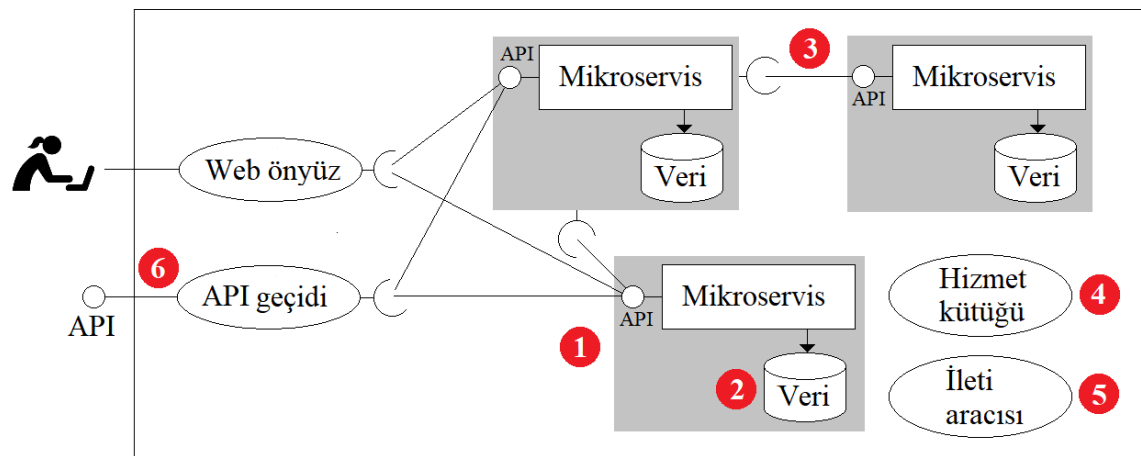
Motosiklet sepeti: Bütün mikroservislerin ihtiyaç duyduğu hizmet keşfi, hata dayanıklı haberleşme gibi kütüphaneleri mikroservislerden ayırıp standartlaştırarak hizmet geliştiricilere yardımcı olmayı amaçlar.

Sunucusuz mimari: Bulut kullanıcılarının yürütüm için gerekli altyapı kaynakları oluşturma ve yönetim problemleri ile uğraşmalarına gerek kalmadan geliştirme, kurulum ve üretime çıkabilmelerini amaçlar.

Hizmet ağı: Motosiklet sepeti teknolojisinin üzerine eklemelerde bulunarak uygulamayı oluşturan mikroservisler arasındaki haberleşme gözleme ve yönetim mekanizmalarına yardımcı olur.

2.2 Mikroservislerin Çalışma Mantığı

Şekil 2.3'de 3 mikroservisten oluşan uygulama diyagramı aracılığıyla mikroservislere ait temel kavramlar tanıtılmaktadır.



Şekil 2.3 Mikroservis mimarisine göre tasarlanmış bir uygulama

Mikroservis yazılım mimarisinde uygulama çok adette mikroservis adı verilen bileşenden meydana gelir (Hoffman, Schnabel & Stanley, 2016) (Richardson, 2019). Bu bileşenlerin her birisi, uygulama tarafından ihtiyaç duyulan fonksiyonlardan bir ve yalnız birini yerine getirir (Şekil 2.2'de: ❶). Yekpare ve hizmet tabanlı mimarinin aksine, fonksiyonların olabildiğince bileşene bölünmesi arzu edilen bir özelliktir.

İdeal bir mikroservis tabanlı uygulama mimarisinde mikroservislerin birbirlerine olan bağımlılıkları en alt seviyede tutulur. Kullandıkları veri katmanı da birbirlerinden yalıtılmıştır: Bütün mikroservislerin serbestçe eriştiği ortak bir veritabanı yerine, her mikroservis kendi ihtiyacı olduğu tablolardaki veriye erişebilir ve bu veriyi denetleyebilir (Şekil 2.2'de: ❷).

Mikroservisler sağladıkları metodları HTTP RESTsel tabanlı bir API ile sunarlar. Bu yöntem, mikroservislerin alternatifi olan SOAP ve servis tabanlı mimarilerin kullanıldığı kurumsal veri yoluna göre daha basittir (Şekil 2.2'de: ❸).

Mikroservisler bilgisayar ağı üzerindeki konumlarını bir hizmet kütüğüne bildirebilirler (Şekil 2.2'de: ❹). Konum bilgisinin merkezileştirilmesi, sistem konfigürasyonunun yönetimini sadeleştirir.

Mikroservisler birbirlerini doğrudan çağırabilecekleri gibi, Kafka (Apache Software Foundation, 2020a) gibi bir ileti aracı uygulaması üzerinden de haberleşebilirler (Şekil 2.2'de: ❺). İleti aracı uygulamasının mimariye entegre edilmesi; yük dağılımı, (bilgisayar ağı problemi gibi nedenlerden dolayı) çağrılarının kaybolmasının önüne geçilmesi gibi avantajlar sağlar.

Uygulamaya dışardan erişim, bir API geçidi üzerinden sağlanır (Şekil 2.2'de: ❻). Bu sayede dışarıya API'nin gerektiği kadar, gerektiği şekilde açılmış olur.

3

DAĞITIK İZLEYİCİLER

Yazılım günlükleri, bilgisayarların kendilerinin ilk günlerinden beri yazılım geliştirmenin ve bakımının ayrılmaz bir ögesi olmuştur. Yazılım sürecine hata ayıklama, güvenlik, işletim ve uygunluk gibi birçok boyutta destek olurlar (Chuvakin, Schmidt, Phillips & Moulder, 2013). Üretim ortamı altındaki bir uygulamanın nasıl ve neden “çalıştığı şekilde çalıştığı”nı anlamak için kullanılan en yaygın yöntemdir. Bununla birlikte, mikroservisler gibi karmaşık yapıdaki modern, dağıtık mimarilerde iki önemli yetersizlikleri göze çarpmaktadır:

- Şekil 3.1’de herhangi bir uygulama tarafından üretilebilecek bir günlük örneği verilmiştir. Karmaşıklığından görülebileceği üzere günlüklerin düz yapısı modern uygulamaların karmaşık mimarisi ve yürütümü için yeterli değildir. Bu durum, günlük analiz işlemini dev metin dosyalarının deşifre edilmesine yönelik yorucu bir dedektiflik çalışması haline getirir.

```
Jan 11 19:12:02 virtualbox-ivan-ubuntu-16 kernel: [ 112.397183] ISO 9660 Extensions: RRIP_1991A
Jan 11 19:12:02 virtualbox-ivan-ubuntu-16 udisksd[3686]: Mounted /dev/sr0 at /media/ivan/vBox_Gst_5.2.2 on b
Jan 11 19:12:02 virtualbox-ivan-ubuntu-16 dbus[742]: [system] Activating service name='org.freedesktop.fwupd
Jan 11 19:12:03 virtualbox-ivan-ubuntu-16 org.freedesktop.fwupd[742]: (fwupd:3801): Fu-WARNING **: Failed to
Jan 11 19:12:03 virtualbox-ivan-ubuntu-16 dbus[742]: [system] Successfully activated service 'org.freedesktop
Jan 11 19:12:12 virtualbox-ivan-ubuntu-16 pulseaudio[3595]: [pulseaudio] bluez5-util.c: GetManagedObjects()
Jan 11 19:12:12 virtualbox-ivan-ubuntu-16 gnome-session[2388]: **: (unity-fallback-mount-helper:3682): WARNING
Jan 11 19:12:12 virtualbox-ivan-ubuntu-16 org.gnome.ScreenSaver[2088]: **: Message: lost the name, shutting d
Jan 11 19:12:22 virtualbox-ivan-ubuntu-16 org.gnome.zentgeist.Engine[2088]: Performing VACUUM operation... 0
Jan 11 19:12:22 virtualbox-ivan-ubuntu-16 org.gnome.zentgeist.Engine[2088]: **: (zeitgeist-database:4005): WAH
Jan 11 19:12:23 virtualbox-ivan-ubuntu-16 com.canonical.Unity.Scope.Applications[2088]: Error loading packag
Jan 11 19:12:23 virtualbox-ivan-ubuntu-16 com.canonical.Unity.Scope.Applications[2088]: (unity-scope-loader:
Jan 11 19:13:05 virtualbox-ivan-ubuntu-16 gnome-session[2388]: /var/lib/dpkg/lock:
Jan 11 19:13:12 virtualbox-ivan-ubuntu-16 dbus[742]: [system] Activating service name='org.debian.apt' (usin
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 AptDaemon: INFO: Initializing daemon
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: 19:13:14 AptDaemon [INFO]: Initializing daemo
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 dbus[742]: [system] Successfully activated service 'org.debian.apt
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 AptDaemon.PackageKit: INFO: Initializing PackageKit compat layer
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: /usr/lib/python3/dist-packages/aptdaemon/work
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: from gi.repository import PackageKitGlib as
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: 19:13:14 AptDaemon.PackageKit [INFO]: Initial
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 AptDaemon: INFO: UpdateCache() was called
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: 19:13:14 AptDaemon [INFO]: UpdateCache() was
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 gnome-session[2388]: (gnome-software:3733): GLib-GObject-CRITICAL
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 AptDaemon.Trans: INFO: Queuing transaction /org/debian/apt/transac
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: 19:13:14 AptDaemon.Trans [INFO]: Queuing tran
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 AptDaemon.Worker: INFO: Simulating trans: /org/debian/apt/transact
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: 19:13:14 AptDaemon.Worker [INFO]: Simulating
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 AptDaemon.Worker: INFO: Processing transaction /org/debian/apt/tra
Jan 11 19:13:14 virtualbox-ivan-ubuntu-16 org.debian.apt[742]: 19:13:14 AptDaemon.Worker [INFO]: Processing
```



Şekil 3.1 Sıradan bir günlük örneği

- Uygulamayı oluşturan otonom bileşenler arasında yürütme bağlam bilgisi paylaşma mekanizmalarının bulunmaması, bu bileşenlere dağılmış uçtan uca bir yürütüm sürecinin tek bir grafikte sunulmasını engeller. Bu bilginin eksikliği yüzünden alt işlem, hangi üst işlemin parçası olduğunu bilemez.

3.1 Günlük Yapısı ve Enstrümantasyon

Günlük oluşturma işleminin gerçekleştirilmesi, enstrümantasyon adlı uygulama kaynak koduna ek kod eklenme adımının yerine getirilmesini gerektirir (Kempf, Karuri & Gao, 2009). Şekil 3.2’de, log4j (Apache Software Foundation, 2020b) adlı yaygın olarak kullanılan bir günlük kütüphanesi ile enstrümantasyonu gerçekleştirilmiş bir kod örneği bulunmaktadır. Örnekte sqlCalistir() metodunun, dugmeTiklandi() metodu altında çalışmasına rağmen, bu önemli hiyerarşik bilginin günlük çıktısına yansımadağı gözlemlenmektedir.

```
private void dugmeTiklandi()
{
    logger.info("dugmeTiklandi");
    int returnedRows = sqlCalistir("SELECT * FROM AYARLAR");
    if (returnedRows == 0)
        logger.info("kayit bulunmadi");
}

private int sqlCalistir(String sql)
{
    logger.info("runSql");
    logger.info("sql: " + sql);
    ...
    return returnedRows;
}

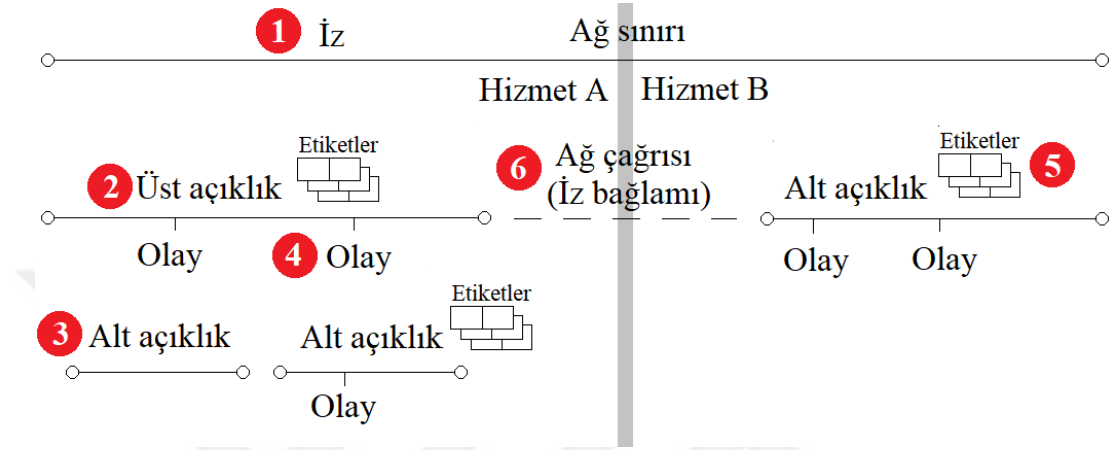
//////////
2018-12-06 19:13:09 INFO Test:11 - dugmeTiklandi
2018-12-06 19:13:09 INFO Test:19 - sqlCalistir
2018-12-06 19:13:09 INFO Test:20 - sql: SELECT * FROM AYARLAR
2018-12-06 19:13:09 INFO Test:14 - kayit bulunmadi
```

Şekil 3.2 Günlükler için örnek enstrümantasyon kodu ve çıktısı

Yürütümün paralel yada dağıtık işlem içermesi durumunda bu düz günlük çıktısını takip etmek ve yorumlamak kısa sürede imkansız hale gelir. Bu problemin önüne geçmek için dağıtık izleyiciler adı altında yeni bir uygulama grubu geliştirilmiştir.

3.2 Dağıtık İzleyici Terminolojisi

Modern dağıtık izleyicilerin kullandığı terminolojinin belirlenmesinde, Google Dapper (Sigelman, vd., 2010) çalışması belirleyici olmuştur. Şekil 3.3’de bu terimler bir diyagram üzerinde gösterilmiştir.



Şekil 3.3 Dağıtık izleyicilerde kullanılan kavramlar

Bir yürütümün uçtan uca ağaç grafiği, iz olarak adlandırılır (Şekil 3.3’de: ❶). Bu iz, hiyerarşik alt açıklıklarına bölünebilir (Şekil 3.3’de: ❷). Bu açıklıkların her birisi, farklı bilgiler içeren üç diziye sahiptir:

Alt açıklıklar: Açıklık tarafından temsil edilen işlem, alt öğelerine bölünebilir (Şekil 3.3’de: ❸). Bu alt açıklıklara denk gelen işlemler, farklı programlama dilleri ve araçları ile geliştirilmiş ve farklı bilgisayarlarda çalışıyor olabilir.

Günlük olayları: Olayın sınıfını (bilgi, ayıklama, uyarı, hata, alarm), olayın oluş zamanını, ve olayın betim bilgisini içerirler (Şekil 3.3’de: ❹). Düz günlük karşılıklarında bulunmayan ‘a açıklığın üst sahibi’ bilgisi sayesinde, bir iz hiyerarşisi içine yerleşirler.

Etiket koleksiyonu: Bir açıklıkla ilişkilendirilmek istenen her türlü bilgiyi içerebilen anahtar-değer çiftleri dizisidir (Şekil 3.3’de: ❺). Anahtarlar (çözüme özel bir işlem no’su gibi) daha sonradan o açıklık ve izi tanımlamak için

kullanılabilecek herhangi bir üstveri olabilir. Açıklığın sunucu yada istemciye mi ait olduğu yada hangi bilgisayarda çalıştığı gibi, açıklığın doğası hakkında bilgi de içerebilirler.

Dağıtık izleyiciler, yürütümün çalışması farklı bilgisayarlara dağılsa bile (Şekil 3.3'de: 6) izi entegre bir şekilde oluşturabilecek ve sunacak görselleştirme araçlarını bünyelerinde barındırırlar.

3.3 Olay Bilgilerinin Toplanması

Dağıtık izleyicilerin temel amacı, bir yazılım uygulamasının çalışma şeklini ve performansını anlamaya yardımcı olmaktır. Bu amaçla o uygulamaya ait olay bilgilerini toplamaya ihtiyaç duyarlar. Bu bilgi toplama üç farklı şekilde yerine getirilebilir:

- Boyut yönelik programlama, işletim sisteminin sağladığı profillemeye altyapısı gibi tekniklerle uygulama kodu üzerinde herhangi bir değişikliğe gerek duymadan olay toplama (Barham, vd., 2003) (Souder, vd., 2001).
- Uygulama kaynak kodu üzerinde değil ama (JVM bytecode gibi) oluşturulmuş makina kodu üzerinde yapılan değişikliklerle olay toplama (Lee & Zorn, 1997)
- Uygulama kodu üzerine elle eklenen özel amaçlı komutlarla (enstrümantasyon) olayların toplanması (Uber, 2020a)(OpenTracing, 2020)

Kod değişikliğine ihtiyaç duymadan olay bilgisi toplamanın dezavantajı oluşturulan izin elle yapılan enstrümantasyona göre ince ayarının iyi yapılamaması, yani toplanacak olayların istenildiği detay seviyesinde belirlenememesidir.

3.4 Dağıtık İzleyicilerde Enstrümantasyon

Dağıtık izleyiciler de görevlerini yerine getirmek için genel günlüklerde olduğu gibi enstrümantasyona ihtiyaç duyar. Genel günlüklerde gerçekleştirilen enstrümantasyondan farkı, işlemlerin hiyerarşik doğasının korunmasıdır.

Şekil 3.4'teki örnek kod OpenTracing (2020) çerçevesine göre enstrümanite edilmiştir. Alttaki sqlCalistir() metodunda oluşturulmuş “sqlCalistir” adlı açıklık, dugmeTiklandi() metodunda oluşturulan “dugmeTiklandi” açıklığının alt-açıklığıdır. Dağıtık izleyicide bu hiyerarşi saklanır ve yürütüm buna uygun olarak görselleştirilir.

```
private void dugmeTiklandi()
{
    try (Scope scope = tracer
        .buildSpan("dugmeTiklandi")
        .startActive(true))
    {
        int returnedRows = sqlCalistir("SELECT * FROM AYARLAR");
        if (returnedRows == 0)
            scope.span().log("kayit bulunmadi");
    }
    catch ()
    {}

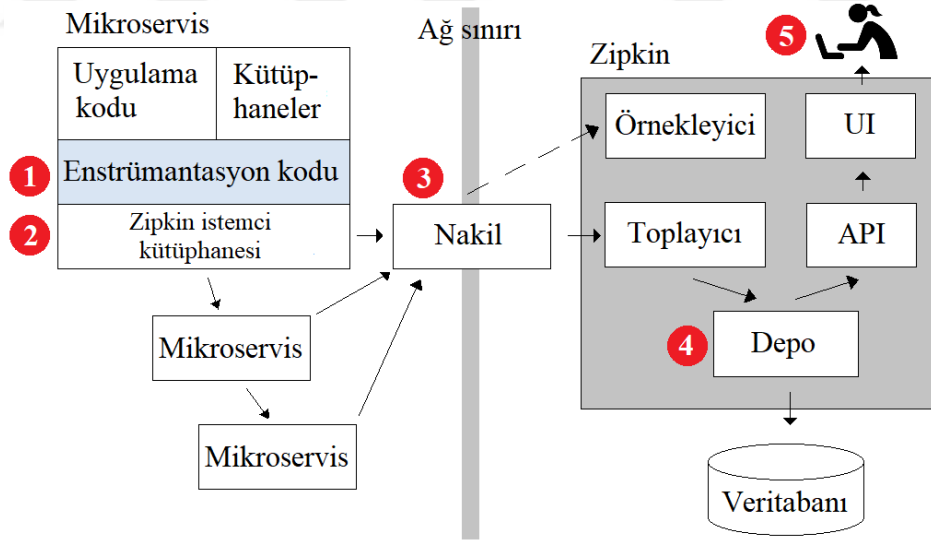
private int sqlCalistir(String sql)
{
    try (Scope scope = tracer
        .buildSpan("sqlCalistir")
        .startActive(true))
    {
        scope.span().setTag("sql", sql);
        ...
        return returnedRows;
    }
    catch ()
    {}
}
```

Şekil 3.4 Dağıtık izleyiciler için örnek enstrümantasyon kodu

Enstrümantasyonun en büyük zorluğu, bütün uygulama kodunu kapsayan ayrı bir planlama ve emek gerektiren bir işlem olmasıdır. Tüm uygulama kodu üzerinde görünür bir varlığa sebep olur.

3.5 Dağıtık İzleyicilerin Çalışma Prensipleri

Şekil 3.5'deki diyagram, popüler bir dağıtık izleyici olan Twitter Zipkin'a (2020) aittir. Dağıtık izleme işleminin yerine getirilmesi için, enstrümantasyon kodunun önceden uygulama koduyla birleştirilmiş olması gerekir (Şekil 3.5'de: ❶). Uygulama yürütümü sırasında, iz günlük olayları uygulamadan dağıtık izleyici istemci kütüphanesine geçer (Şekil 3.5'de: ❷). Tüm bileşenlerce oluşturulan bu olaylar bir ağ taşıma mekanizması ile dağıtık izleyici sunucusuna nakledilirler (Şekil 3.5'de: ❸). Toplanan tüm olaylar daha sonra analiz edilebilmeleri için dağıtık izleyici sunucusunun veritabanında depolanır (Şekil 3.5'de: ❹). Bu olaylar izleyicinin kullanıcı arabirimi aracılığıyla analiz edilebilirler (Şekil 3.5'de: ❺).



Şekil 3.5 Twitter Zipkin dağıtık izleyicisine ait diyagram

Tablo 3.1 Dağıtık izleyiciler ve kullanım amaçları

	Yayımlanma Tarihi	Arıza tespiti	Sürekli durum hataları tespiti	Dağıtık profilleme	Kaynakların kullanımı	İşyükü modelleme
ETE (Hellerstein vd., 1999)	Haziran 1999			●		●
Pinpoint (Chen vd., 2002)	Haziran 2002	●	●			●
Magpie (Barham vd., 2003)	Mayıs 2003	●				●
Stardust (Thereska vd., 2006)	Haziran 2006				●	●
Whodunit (Chanda, Cox & Zwaenepoel, 2007)	Mart 2007			●		●
X-Trace (Fonseca vd., 2007)	Nisan 2007					●
Quanto (Fonseca, Dutta, Levis & Stoica, 2008)	Aralık 2008		●			●
Dapper (Sigelman vd., 2010)	Nisan 2010		●	●		●

Carnegie Mellon Üniversitesi Paralel Veri Laboratuvarı, dağıtık izleyiciler için beş ana kullanım senaryosu belirlemiştir (Sambasivan, vd., 2014):

Arıza Tespiti: Koddaki yerel yazılım böceklerinin tespiti

Sürekli Durum Hataları Tespiti: Performans darboğazları, kaynak kullanım problemleri gibi mimariyle yada altyapıyla ilgili problemlerin tespiti

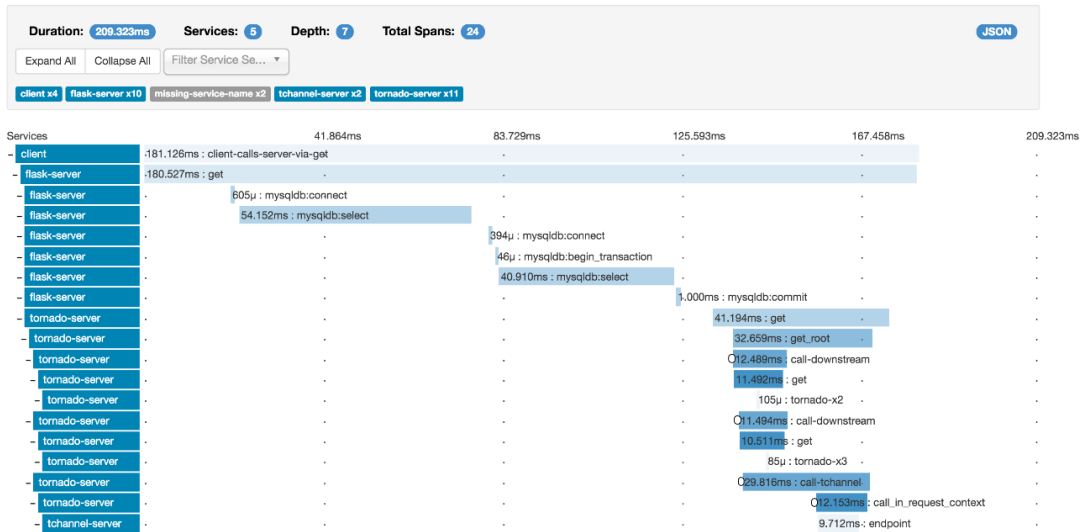
Dağıtık profilleme: Normal bir uygulama yürütüm profilleme işleminin, dağıtık bir çözüme uygulanması

Kaynakların kullanımı: Uçtan uca yürütüm içerisinde, yürütülen bir işlem tarafından kullanılan kaynakların (işlemci zamanı, ağ trafiği) kime ait olduğunun/kime faturalandırılacağıın tespiti

İşyükü modelleme: Uçtan uca yürütüm analizi için oluşturulan iz ağacının oluşturulması için gerekli işlemler

3.6 Dağıtık İzleyicilerin Kullanıcı Arayüzleri

Şekil 3.6'de Twitter Zipkin (2020), Şekil 3.7'da Uber Jaeger (2020a) dağıtık izleyicilerinin kullanıcı arabirimine ait ekran görüntüleri bulunmaktadır. İki dağıtık izleyici görselleştirme bakımından birbirlerine yakın bir yol izler ve x-ekseninde zamanın temsil edildiği, y-ekseninde izin ve alt açıklıkların görüntülediği Gantt şemasını kullanırlar.



Şekil 3.6 Twitter Zipkin dağıtık izleyici

3.7 Örnekleyiciler

Yoğun kaynak gerektirdiği için üretim ortamında oluşan her bir günlük olayının saklanması pratik değildir. Bu amaçla, örnekleyiciler hangi günlük olay alt kümesinin saklanması gerektiği ve yeterli olacağına dair çeşitli algoritmalar içerir. Örneğin Jaeger (Uber, 2020b)'da şu örnekleme stratejilerini sunar:

Sabit Örnekleyici: Bütün izler için aynı stratejiyi uygular; parametre değerine göre ya hepsini örnekler yada hiç birisini örneklemez.

Olasılık: Parametre değeriyle belirtilen oranda iz örneklenir. Örneğin 0.1 değeriyle, 10 izden biri örneklenir.

Oran sınırlı: Örneklemenin sabit bir oranda yapılması sağlanır. Örneğin 2.0 değeri kullanıldığında, saniyede 2 izin örneklenmesine sebep olur.

Uzaktan Örnekleyici: Kullanılacak stratejiyi, Uber Jaeger etmenine sorarak edinir. Böylece örnekleme stratejisi merkezi bir şekilde ayarlanabilir.

Bir iz için örnekleme yapılıp yapılmayacağı, bir izin başında belirlenir. Örneklemeye için seçilen bir izin bütün alt açıklıkları örneklemeye dahil edilir.

3.8 Yayılım

Yürütüm bağlam bilgisinin bir bileşenden diğerine nakledilmesi için gerekli mekanizmalar yayılım olarak adlandırılır. İzleyiciler iki çeşit yayılıma ihtiyaç duyar:

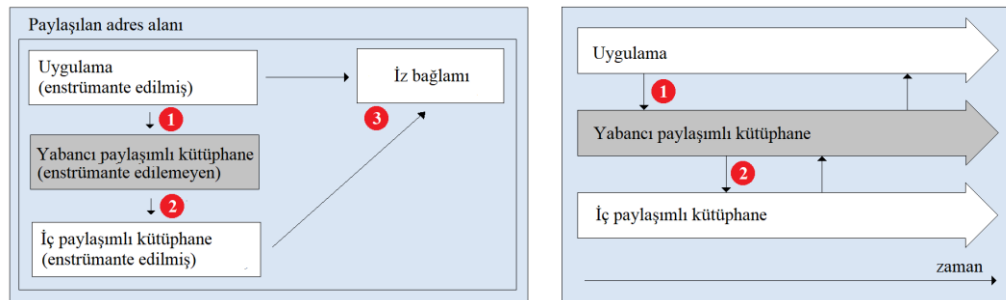
Bilgisayar ağı yayılımı: Bu yayılım çeşidi, bağlam bilgisinin ayrı süreçlerde yada bilgisayar ağı içinde paylaşılmasıyla ilgilidir. Mikroservis mimarisinde tercih edilen haberleşme mekanizması HTTP çağrılarına dayandığı için, bağlam bilgisinin naklinde HTTP üstbilgilerinin kullanımı yaygın ve pratik bir yöntemdir. Üstbilgilerin bu amaçla ilk kullanımı X-Trace dağıtık izleyicisi (Fonseca vd., 2007) ile gerçekleştirilmiştir.



Şekil 3.9 Bilgisayar ağı yayılımı

Bilgi paylaşımı için, çağrıyı yapan tarafın HTTP çağrısına iki üst bilgi alanını içitmesi yeterlidir. Şekil 3.9’da, “spanid” üst açıklık no’ya, “traceid” ise bu işlemin ait olduğu ana ize ait tekil tanımlayıcıdır. Çağrıyı yapan tarafın içittiği bu değerleri, çağrı yapılan taraf çeker. Böylece hangi işlemin parçası olduğunu anlar.

İşlem içi yayılım: Dağıtık izleme geliştiricilerinin, bilgisayar ağı yayılımının dışında gözönünde bulundurmaları gereken ikinci bir yayılım tipi vardır.



Şekil 3.10 İşlem içi yayılım

Sınama altındaki uygulamamızın, müdahale imkanımızın olmadığı bir yabancı paylaşımlı kütüphaneyi çağırdığı (Şekil 3.10’da: ❶), dış kütüphanenin de bizim tarafımızdan geliştirilmiş diğer bir iç paylaşımlı kütüphaneyi çağırdığı (Şekil 3.10’da: ❷) bir senaryo düşünelim: Bu durum, uygulama yazılımlarını hizmet sağlayıcı paylaşımlı kütüphanelerden bağımsız hale getirmek için yaygın olarak

kullanılan bir denetim evirme tasarım örüntüsüdür ve Microsoft WOSA'dan TWAIN tarayıcı kütüphanesine, Java Advanced Imaging kütüphanesine birçok farklı uygulamada kullanılır. Birazdan bahsedilecek OpenTracing çerçevesi de bu örüntüyü kullanır.

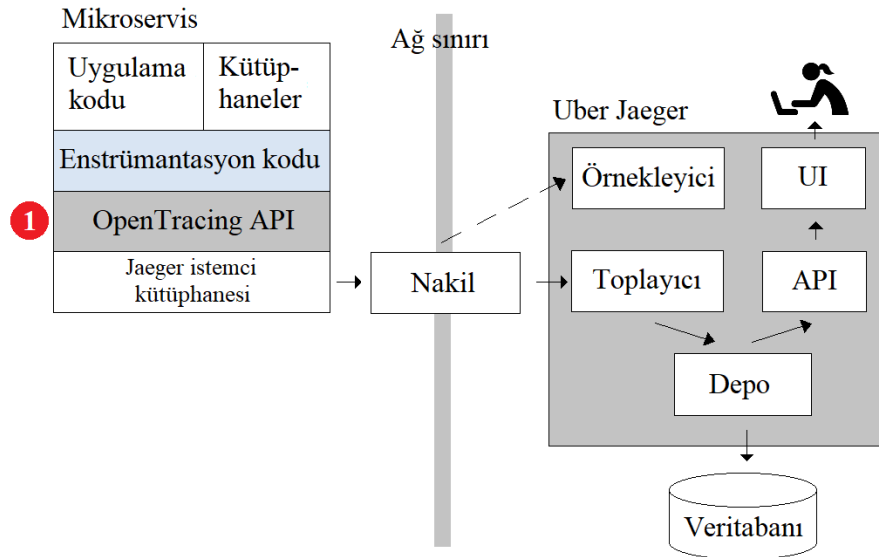
Çağrı bağlam bilgisini kendi uygulamamızdan iç kütüphanemize nasıl nakledebiliriz ki iz ağacımızı oluşturabilelim? Dağıtık izleyiciler, bu problemi yayılım bilgisini iş parçacığı yerel hafızası veya benzer ortak bir saklama alanı kullanarak çözme yoluna giderler (Şekil 3.10'da: ③).

3.9 Standartlaştırma Çalışmaları

Kapsamlı bir enstrümantasyon, uygulamanın bir kısmında değil kodun tamamını kapsayacak şekilde gerçekleştirilebilir. Bu sebeple büyük çaplı bir uygulamada bir izleyiciden diğerine geçiş büyük emek gerektirir ve satıcı bağımlılığına sebep olur. Bu problemin önüne geçmek için farklı çalışma grupları standartlaştırma çalışmaları yürütmektedir:

- **W3 Dağıtık İzleyici Çalışma Grubu** bilgisayar ağı yayılımında kullanılan HTTP üstbilgilerinin standartlaştırılmasıyla ilgilenir. Bu amaçla bir izleyici bağlam önerisi yayınlamıştır (World Wide Web Consortium, 2018). Öneri, izleyici bağlam bilgisi paylaşımının mahremiyet ve güvenlik boyutları hakkında tavsiyeler de içermektedir.
- Hem **OpenTracing** (2020), hem **OpenCensus** (2020) çözüm sağlayıcıdan bağımsız dağıtık izleyici çerçeveleri oluşturmayı, bu sayede yapılmış bir enstrümantasyonun aynı anda farklı dağıtık izleyiciler ile uyumlu olmasını amaçlar. OpenCensus izleyici fonksiyonlitesine ek olarak, ölçüm toplama işlemini ve ağ yayılımını standartlaştırmayı da amaçlar.

OpenTracing ve OpenCensus Mayıs 2019'da **OpenTelemetry** (2020) adı altında birleşme kararı almıştır.



Şekil 3.11 OpenTracing destekli Jaeger dağıtık izleyicisine ait diyagram

Şekil 3.11'deki diyagramda OpenTracing (2020) uyumlu bir dağıtık izleyici olan Uber Jaeger (2020a) uygulamasının ana bileşenleri gösterilmektedir. Enstrümantasyon koduyla istemci kütüphanesi arasına Şekil 3.5'da bulunmayan OpenTracing API katmanı eklenmiştir (Şekil 3.11'de: ❶). Bu eklemeyeyle sağlanan soyutlama sayesinde, yazılım geliştiricileri ve sistem yöneticileri herhangi bir kod değişikliğine gerek kalmadan ve çok az bir emek harcayarak ihtiyaç duyduklarında bir izleyiciden başka bir izleyiciye geçiş yapabilirler.

Günümüz yazılım sistemleri insanlığın oluşturduđu en karmaşık yapılarıdır. Bu nedenle yazılımın oluşturulması, bakımı, anlaşılması ve öğrenilmesi çaba gerektirir. Yazılım görselleştirme; yazılımı yapısı, yürütümü ve evrimi gibi farklı boyutlarını anlaşılır bir şekilde resmederek bu işlemlere yardımcı olma amacındadır.

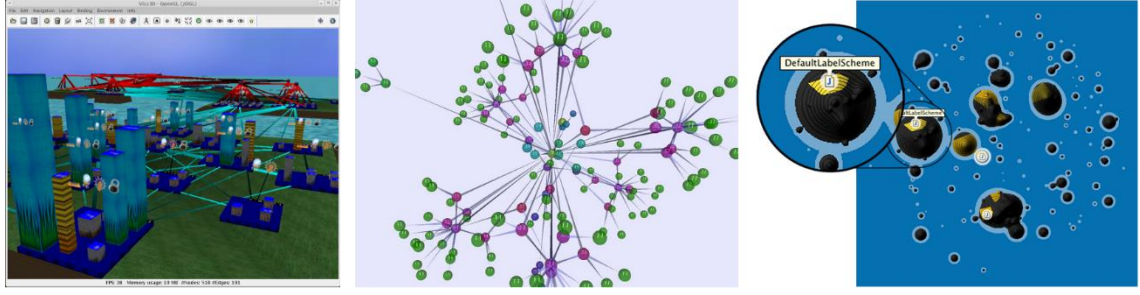
Diehl (2007, s. 3), yazılım görselleştirme çalışmasını üç ana gruba ayırır:

Yapısal: Uygulama kaynak kodu statik bir metin olarak ele alınır. Metin, uygulamanın çalıştırılmasına gerek kalmadan analiz edilerek uygulamayı oluşturan veri yapıları ve bu yapıların birbirleriyle ilişkileri görselleştirilir.

Davranışsal: Uygulama, verilen gerçek yada soyut girdiler ile çalıştırılır. Uygulamanın yürütümünde hangi komutların çalıştırıldığı ve program durumunun değişimi görselleştirilir.

Evrimsel: Geçen zaman içinde bir uygulamanın kullanıcılarının değişen ihtiyaçlarını karşılayacak şekilde güncellenmesi gerekir. Eğer başarılı bir uygulama geliştirilmezse müşterilerini rakiplerine kaptırır. Evrimsel görselleştirme uygulamanın değişmekte olan tasarım ve konfigürasyon ihtiyaçlarını görselleştirmeyi amaçlar.

Tez çalışması boyunca Şekil 4.1’de görüldüğü gibi uygulamanın yürütümünü şehir bloklarından molekül yapılarına, haritalara farklı metaforlarla görselleştirerek, uygulamanın anlaşılabilirliğine katkıda bulunma iddiasında olan çok farklı ve çok yaratıcı çalışmalarla karşılaştık. Bu çalışmaların çeşit ve adetlerinin yüksekliğinden dolayı, incelememizi XPLOA benzeri dinamik görselleştirme için sıralama diyagramı kullanan araçlarla sınırlandırdık.

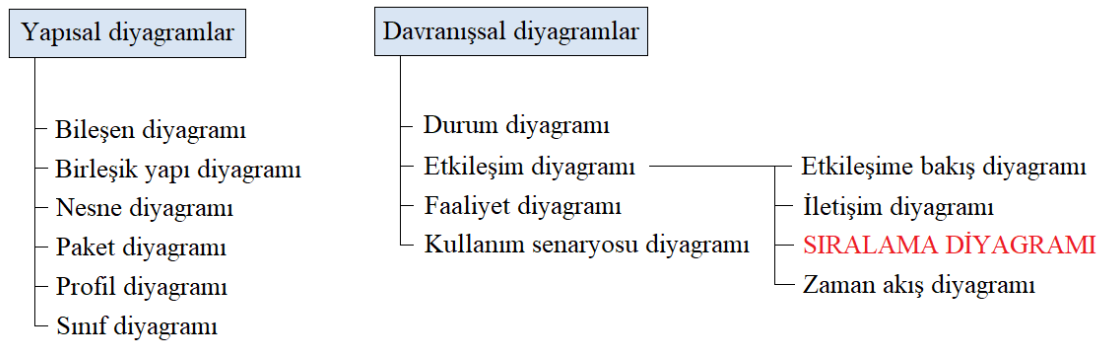


Şekil 4.1 Kodun şehir, molekül ve takımada metaforlarıyla görselleştirilmesi

XPLORA aracının temel hedefi, yazılımın dinamik yürütümünü kolay anlaşılır bir şekilde görselleştirmektir. Görselleştirme mekanizması olarak UML 2.0 sıralama diyagramları tercih edildiği için öncelikle bu diyagram tipi tanıtılmıştır. Daha sonra sıralama diyagramlarının alternatiflerine kısaca değinilmiştir.

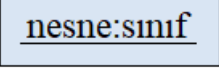
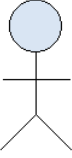
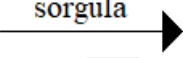
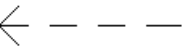
4.1 Sıralama Diyagramları

UML davranış diyagramları altında sınıflandırılan sıralama diyagramları, bileşenler arasındaki etkileşimi ve bilgi transferini görselleştirmek için hem sistem tasarımcıları hem iş analistleri tarafından yaygın olarak kullanılan bir diyagram çeşididir (Unhelkar, 2018, s. 25). Şekil 4.2’de sıralama diyagramlarının tüm UML diyagramları arasındaki konumu gösterilmiştir.



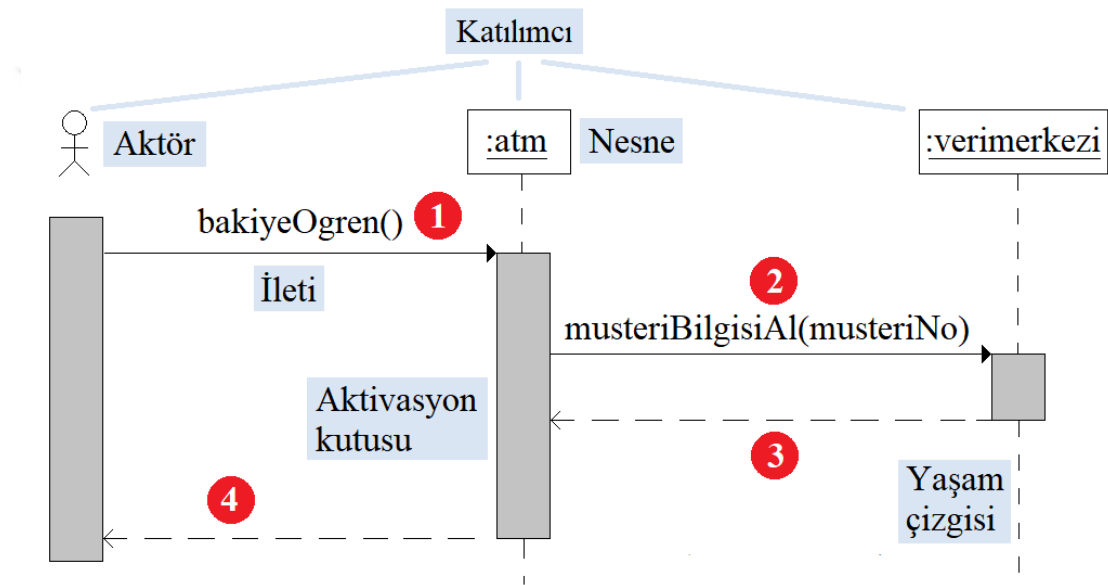
Şekil 4.2 Sıralama diyagramlarının UML diyagramları arasındaki yeri

Tablo 4.1 Sıralama diyagramı öğeleri

	Nesne¹: Sistemin içinde bulunan ve diğer nesnelerle iletişim halinde olan nesnelerdir. ‘birNesne:Sınıf’ ifadesinde iki noktadan önceki kısım değişken adını, sonraki kısım nesnenin ait olduğu sınıfı ifade eder.
	Aktör: Aktörler, sistemle etkileşim halinde olmakla birlikte inşa edilen sistemin kapsamının dışında kalan sistem yada kişilerdir. Örneğin sistemin bir kullanıcısı bu şekilde gösterilir.
	Yaşam çizgisi: Her bir paralel düşey çizgi, farklı bir katılımcıyı sembolize eder.
	Etkinleşme kutusu: Yaşam çizgilerinin üzerine çizilen dikdörtgenler olarak gösterilir. Bu dikdörtgenler, ileti sonucu gerçekleşen süreci gösterir.
	İleti: Nesneler arasındaki etkileşimler, ileti adı verilen oklarla gösterilir. Okun üzerinde iletinin adı bulunur. Senkron çağrılar, okların başının dolu olması ile gösterilir.
	Asenkron ileti: Açık ok başları, asenkron ileti çağrılarını gösterir.
	Öz ileti: Nesneler kendi metotlarını da çağırabilir. Bu durumda, etkinleşme kutusunun işlem seviyesi bir artmış olur.
	Dönüş iletisi: Asenkron çağrılarının sonunda, çağrılan taraf görevini tamamladığında çağrıyı yapan tarafa geri dönüşü sağlar.
	Yokedici: Nesne yok edildiğinde (bellekten kaldırıldığında), yaşam çizgisinin en altına X işareti çizilir ve yaşam çizgisi sonlandırılır.

¹ Katılımcılar, nesneler ve aktörler olmak üzere iki sınıfa ayrılır.

Sıralama diyagramlarında nesneler arasındaki etkileşim, zamana göre sıralanmış olarak temsil edilir. Zaman y-ekseninde gösterilirken, dahil olan katılımcılar ve etkileşimleri x-ekseninde gösterilirler. Sıralama diyagramınca modellenen senaryodaki nesnenin yaşam süresi, yaşam çizgisi ile temsil edilir. Etkinleşme kutusu işlemin yürütüldüğü süreyi temsil eder. Alt etkileşim kutusunu etkinleştirme çağrısı, okun başı alt etkileşim kutusunu işaret edecek şekilde bir okla temsil edilir. Tablo 4.1’de sıralama diyagramlarının temel öğeleri listelenmiştir.



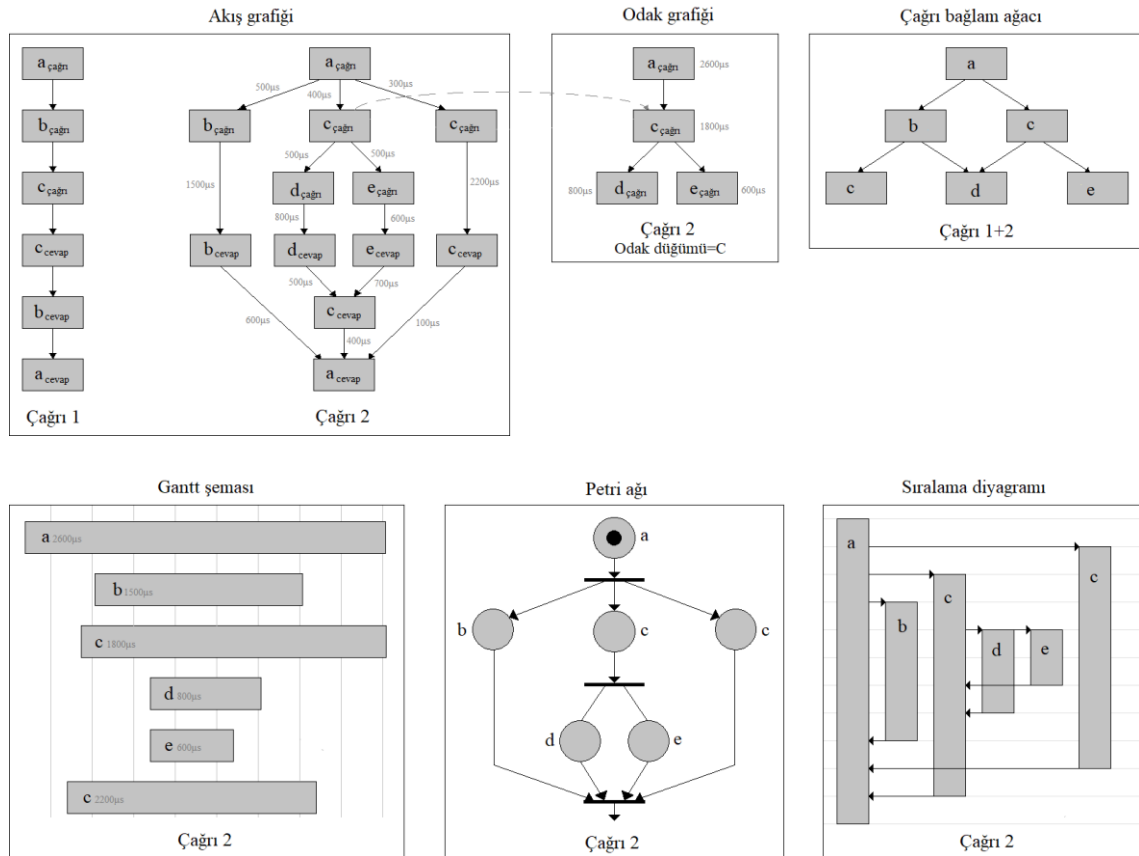
Şekil 4.3 Örnek bir sıralama diyagramı

Şekil 4.3’deki örnek sıralama diyagramında müşteri aktörü, bakiyesini öğrenme talebiyle atm nesnesine bakiyeOgren() senkron iletisini göndererek süreci başlatır (Şekil 4.3’de: 1). atm nesnesi, verimerkezi nesnesinin istemcisi durumundadır. İşlemi talep eden müşterinin hesap bilgilerini edinmek için verimerkezi nesnesine senkron müşteriBilgisiAl() iletisini gönderir (Şekil 4.3’de: 2). Bilgiyi edindikten sonra (Şekil 4.3’de: 3), müşteri aktörüne geri döndürür (Şekil 4.3’de: 4).

4.2 Alternatif İz Görselleştirme Yöntemleri

Dağıtık izleyiciler tarafından kullanılan Gantt şemaları ve XPLORA aracı tarafından kullanılan sıralama diyagramları, yazılım akışının görselleştirilmesinde kullanılabilecek grafik tiplerinde yalnızca iki tanesidir. Bu bölümde, bir yürütümün akışını görselleştiren alternatif grafik tiplerinin her birine kısaca değinilmiştir.

Şekil 4.4’de Sambasivan vd.’nin (2014, s. 15) çalışmasına yeni grafik tipleri eklenerek aynı akışın görselleştirilmesi karşılaştırılmıştır.



Şekil 4.4 Alternatif yürütüm görselleştirme yöntemleri

4.2.1 Akış Grafiği

Yönlü döngüsüz yapıdadırlar. Genelde aynı akışa sahip birden fazla isteğin bir toplam özetini görselleştirme amacıyla kullanılırlar. Grafikteki çatallar eşzamanlı

aktivitenin başlangıcını, birleşimler eşzamanlama noktalarını gösterir. Stardust (Thereska vd., 2006) ve X-Trace (Fonseca vd., 2007), izleri akış grafikleri şeklinde görselleştirir.

4.2.2 Odak Grafiği

Bu görselleştirme tipi de çoklu izleri beraber göstermek için kullanılır. Ancak eşzamanlılık, çatallanma veya birleşmeleri gösteremez. Odak grafikleri, bir ana işlev tarafından erişilen işlevleri göstermek için yelpaze çıkışlarını kullanır. "Odak düğümü" olarak seçilmiş bir bileşen yada fonksiyonun çağrı yığını ve grafiğini gösterir. Odak grafiklerinin en faydalı olduğu durumlar, geliştiricilerin problemleri fonksiyonu yada bileşeni zaten bildikleri ve ek analize ihtiyaç duydukları durumlardır. Dapper (Sigelman vd., 2010), aynı iş akışına sahip birden fazla isteği göstermek için odak grafiklerini kullanır. Farklı iş akışlarına sahip birden fazla isteği görselleştirmek için kullanıldığında, kullanılmamış yolları da göstermesi yanlış değerlendirmelere neden olabilir.

4.2.3 Çağrı Bağlam Ağacı

Farklı akışa sahip birden fazla isteği bir arada göstermek için kullanılır. En tepeden uca dağıtık sistemdeki geçerli her alt çağrıyı gösterir. Çağrı bağlam ağaçları, sabit bir zaman aralığı için oluşturulabilir ve en üst düzeyde sistem davranışının özetinin gerektiği (profil oluşturma gibi) durumlarda kullanılır. Whodunit (Chanda vd., 2007), iş yükleri için profil bilgilerini göstermek için çağrı bağlam ağaçlarını kullanır.

4.2.4 Gantt Şeması

Genelde bireysel akışları görselleştirmek için kullanılır. Ancak akışlar aynı yapıdaysa, birden fazla istek için de kullanılabilir. y-ekseni, dağıtık sistem tarafından yapılan genel istek ve bu çağrı sonucu oluşan alt istekleri gösterir. x-ekseni ise zamanı gösterir. Eşzamanlılık, x-ekseninde aynı değere denk gelen çubukların görsel olarak tanımlanmasıyla kolaylıkla anlaşılabilir. Çatallar ve bağlantılar da görsel olarak tanımlanabilir, ancak bu işlem eşzamanlılığın tespitine göre daha zordur. ETE (Hellerstein vd., 1999) ve Dapper (Sigelman vd., 2010), bireysel izleri görselleştirmek için Gantt şemalarını kullanır. Genel istek ve alt istek

gecikmelerinin gösterilmesine ek olarak, Dapper sunucuda harcanan süreyi, isteğin veya alt isteğin gözlenen gecikmesinden çıkartarak ağda geçen süresini de belirler.

4.2.5 Petri Ağı

Petri ağları, 1962 yılında Carl Adam Petri tarafından doktora tez konusu olarak geliştirilmiştir (Reisig, 2013). O günden beri kavramları genişletilmeye ve geliştirilmeye devam etmektedir. Ofis otomasyonu, iş akışları, esnek üretim, programlama dilleri, protokoller ve ağlar, donanım yapıları, gerçek zamanlı sistemler, performans değerlendirme, işlem araştırması, gömülü sistemler, savunma sistemleri, telekomünikasyon, internet, e-ticaret ve ticaret, demiryolu ağları, biyolojik sistemler Petri ağlarının uygulandığı alanlardan bazılarıdır.

Petri ağları, konumları ve geçişleri birbirine bağlayan yönlendirilmiş arklardan oluşan bir dizidir. Konumlar belirteç tutabilir. Bir ağın durumu veya işareti, belirteçlerin yerlere atanmasıdır. Arkların varsayılan kapasitesi 1'dir; 1'den büyükse, kapasite ark üzerinde işaretlenir. Konumlar sonsuz kapasiteye sahiptir. Geçişler kapasiteye sahip değildir ve belirteç depolayamaz. Girdi konumlarının her birindeki belirteç sayısı, konumdan geçişe giden ark ağırlığına eşit olduğunda geçiş etkinleşir. Etkinleştirilmiş bir geçiş herhangi bir zamanda ateşlenebilir. Ateşleme işlemiyle, giriş konumlarındaki belirteçler, ark ağırlıkları ve konum kapasitelerine göre çıkış konumlarına taşınır. Ağ, belirteç dağılımının değişmesiyle yeni bir duruma ulaşmış olur.

Bir geçiş ateşlendiğinde, giriş konumlarından kendisini etkinleştiren belirteçleri alır ve bu belirteçleri ark ağırlıklarına göre çıkış belirteçlerine dağıtır. Ark ağırlıkları tamamen aynıysa, belirteçler geçiş boyunca hareket etmiş olurlar. Ancak, farklılarsa varolan belirteçler yok olabilir yada yeni belirteçler oluşabilir.

Her mühendislik etkinliğinde olduğu gibi yazılım sına da denetlenebilirliği ve tekrar edilebilirliği sağlama amacındadır. Bu nedenle betikleştirilmiş sına kabul gören sına tipi olmuştur. Ama tespit edilmesi zor hatalar ve yazılım böceklerinin düzensiz doğası, sinamanın bir grup otomatikleşmiş adımın çalıştırılmasından öte bir düşünce faaliyeti olmasını zorunlu kılar. Bir betiğin otomatik ve tekrarlanan yürütümleri, sinamanın “böcek ilacı paradoksu”ndan¹ dolayı kısa sürede etkinliğini kaybetmesine sebep olur (Graham, Van Veenendaal, Evans & Black, 2008, s. 19).

Betik kullanımının makineleşmiş doğasına tepki olarak bireysel sinayıcının işini en sağlıklı şekilde yürütebileceği, kişisel özgürlüğünü, bilgi birikimini ve sorumluluğunu vurgulayan bir yazılım sına yöntemi olan keşif sinaması popülerlik kazanmıştır. İki sına tipinin yaklaşımlarındaki fark, bir karikatüristin gözünden Şekil 5.1’deki gibi resmedilmiştir (Our Sky, 2018).



Şekil 5.1 Keşif sinamasının betikleştirilmiş sinamayla karşılaştırılması

¹ Böcek ilacı paradoksu: Gerçek bir böcek türünün bir böcek ilacına karşı bağışıklık kazanması gibi, belli sına senaryoları için hazırlanmış sabit betikler zamanla etkisini kaybeder. Yani yeni yazılım böceklerini yakalayamaz.

Şekil 5.2, ISTQB'nin temel sınam teknikleri müfredatını içermektedir. Keşif sınaması bu müfredatta kendisine yer bulmuştur (Graham, vd. 2008, s. 86).

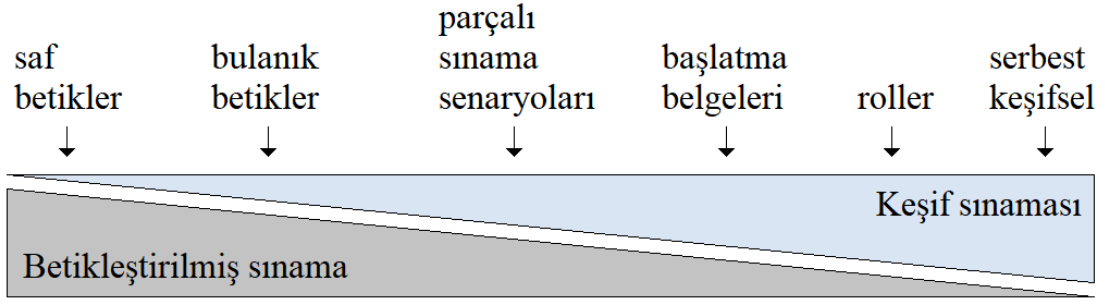


Şekil 5.2 Keşif sınamasının ISTQB eğitim müfredatındaki içindeki yeri

Geleneksel betikleştirilmiş sınamada öne çıkan denetlenebilirlik ve tekrar edilebilirlik, keşif sınamasında yerini adaptasyon yeteneği ve öğrenmeye bırakır. Önceden planlama ve otomatikleştirilmiş sınamanın temellerini oluşturduğu betikleştirilmiş sınamanın aksine, keşif sınamasının planlama, yürütüm ve değerlendirmenin eş zamanlı yürütüldüğü bir süreç olarak değerlendirilir (Whittaker, 2009).

Keşif sınamasında yaygın olarak karşılaşılan bir yanlış anlama, önceden planlama vurgulanmadığı için düzensiz, kuralsız ve ciddiyetsiz bir şekilde yerine getirilen bir sınam tipi olarak değerlendirilmesi ve doğaçlama sınam ile karıştırılmasıdır (Bach, 2006). Keşif sınamasında sınamalar genelde 45-90 dakikalık oturumlar halinde yapılır. Bu oturumların amacını ve kapsamını net bir şekilde belirlemek için de başlatma belgeleri adlı dokümanlar kullanılır.

Keşif sınaması yararlı ve etkili bir araç olarak kabul edilse de, yapısal ve sistemli sınam tekniklerinin tamamlayıcısı olduğu gözden kaçırılmamalıdır. Betikleştirilmiş sınam ve keşif sınaması birbirinden kopuk değildir: Şekil 5.3'de aralarındaki sürekliliği sağlayan seviyeler gösterilmiştir (Kaner, 2004, s. 14).



Şekil 5.3 Betikleştirilmiş sinama – Keşif sinaması sürekliliği

Saf betikler: Sinama tamamen otomatikleştirilmiş ve tüm detayları netleştirilmiş betikler üzerinden gerçekleştirilir.

Bulanık betikler: Sinama adım adım belirlenmiştir, ama belirtilmesi kesinlikle zorunlu olmayan detaylar belirtilmez.

Parçalı sinama senaryoları: Sınamaya konu olacak alanlar, kullanım senaryosu yapısındaki bir yada iki cümle ile tarif edilir.

Başlatma belgeleri: 90 dakika civarındaki test oturumunun hedefinin ve yürütümünün kısaca tarif edildiği dokümanlardır.

Roller: Sınayıcı, ürünü belli bir açıdan test etmekle görevlendirilir.

Serbest keşifsel: Keşifsel sinama tekniklerinin henüz netleşmediği 90'lı yıllar dönemini kapsar. Sınayıcı bilgi, tecrübe ve sezgilerine dayanarak tüm sinama sürecini belirler.

5.1 Keşif Sınamasının Özellikleri

Itkonen ve Rautiainen (2005, s. 86-87), keşif sınamasını beş özellikte özetler:

- Sınamalar, önden detaylandırılmış sinama betikleri veya sinama senaryoları olarak tanımlanmazlar. Keşif sinaması, adım adım takip edilecek talimatlar içermeyen bir keşif çalışmasıdır.
- Keşif sinamasına, önceden gerçekleştirilmiş sınamalar ve bu sınamalardan elde edilen sonuçlar rehberlik eder. Keşif sınavıcısı sınanacak şeye ait

gereksinim dokümanı, kullanım kılavuzu, hatta bir pazarlama broşürüne kadar her türlü bilgiden yararlanır.

- Keşif sınavının amacı hataları detaylı sına senaryoları oluşturarak bulmak yerine, keşif yöntemiyle bulmaktır.
- Keşif sınavı, sına altındaki sistemin öğrenilmesi, sına tasarımı ve sına yürütümünün eş zamanlı olarak yerine getirilmesidir.
- Sınanın etkinliği sınavıcının bilgi, yetenek ve tecrübesine bağlıdır.

5.2 Keşif Sınavının Temel İhtiyaçları

Yazılım sınavının temel zorlukları, keşifsel sına için de geçerlidir. Kaner ve Bach (2004, s. 16), ilgili konu başlıklarını şu şekilde özetlemiştir:

Öğrenme: Programı nasıl öğrenebiliriz?

Görünürlük: Yüzeyin altını nasıl görebiliriz?

Denetim: İç veri değerlerini nasıl değiştirebiliriz?

Risk: Çalıştırılacak en iyi testler hangileridir?

Yürütüm: Testlerin çalıştırılabileceği en verimli yol hangisidir?

Lojistik: Test yürütümü için gerekli ortam nedir?

Uzman problemi: Test sonucunun doğru olduğunu nasıl söyleyebiliriz?

Bildirim: Başarısız işlemi nasıl tekrarlayabiliriz ve etkin bir şekilde bildirebiliriz?

Dokümantasyon: Hangi test dokümantasyonuna ihtiyacımız var?

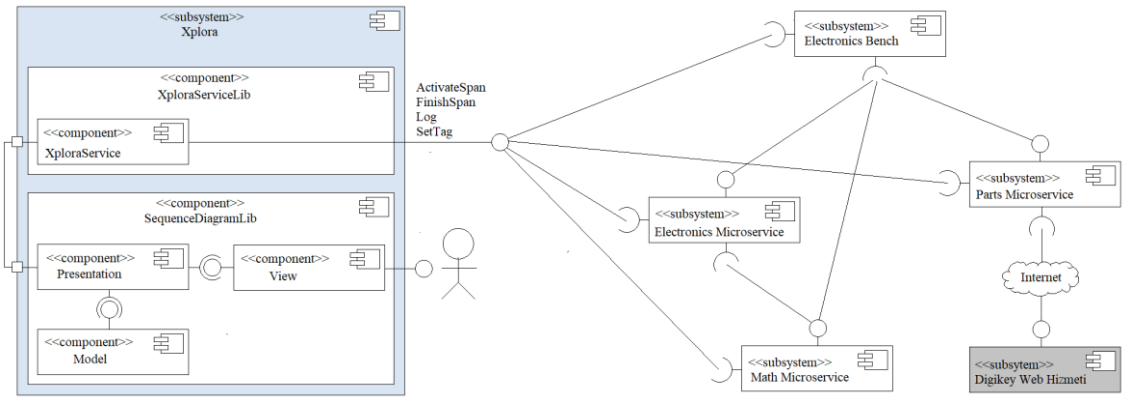
Ölçüm: Uygun olan ölçümler hangileridir?

Durma: Testin ne zaman durdurulacağına nasıl karar verilir?

Eğitim ve Denetim: Sınavıcıların etkin olmalarına nasıl yardım edilebilir.

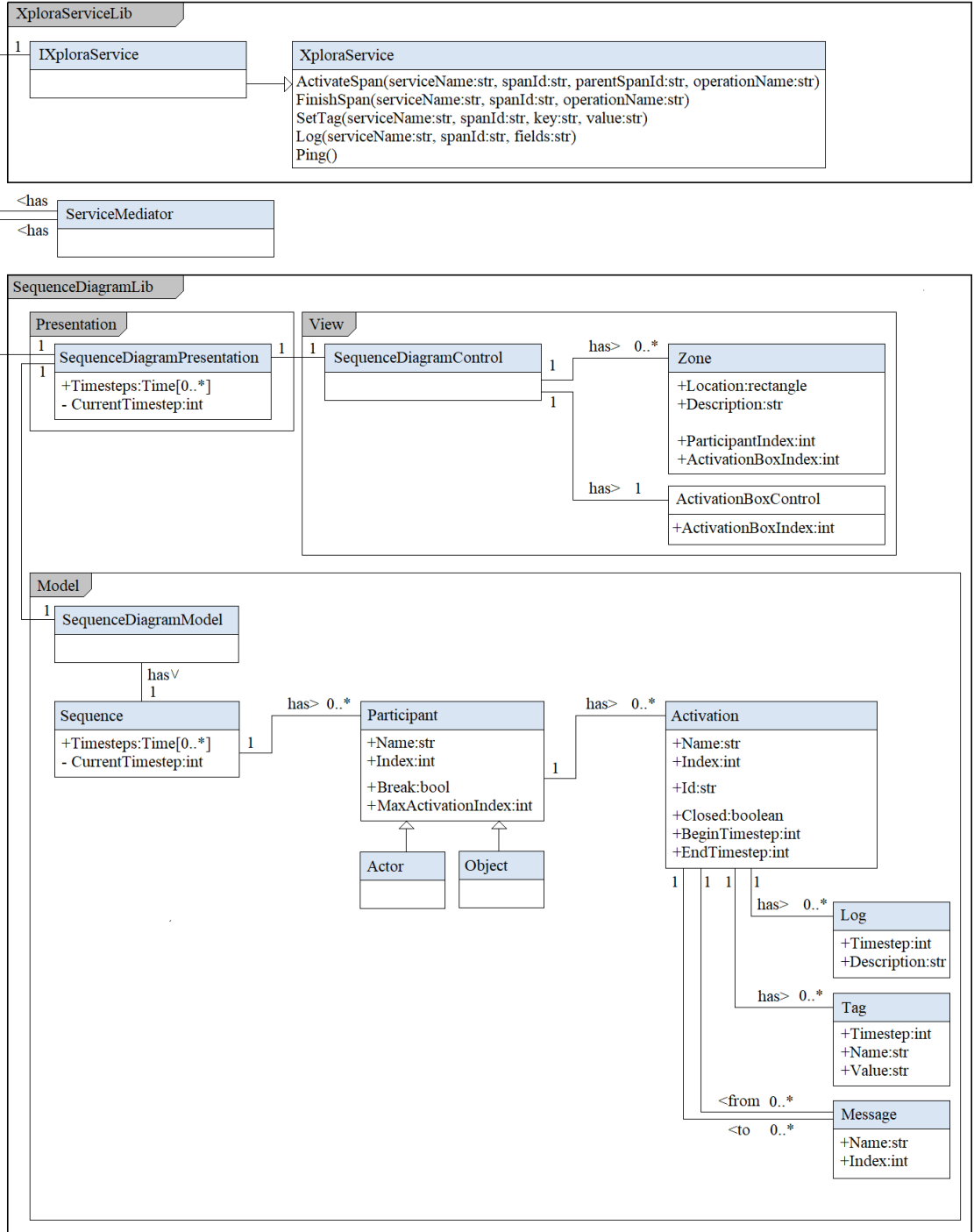
Bu ihtiyaçlara yardımcı olacak ve uygulamanın iç yapısını anlaşılması kolay bir şekilde sunacak bir sına aracı, keşif sınavıcısı için büyük yarar sağlayacaktır.

XPLORA keşif aracı, Visual C# ile geliştirilmiş bir .NET uygulamasıdır. Araç, OpenTracing API'si ile enstrümente edilmiş diğer uygulamalardan gelen iz olaylarını RESTful web servisi aracılığıyla toplar ve bu olayları bir sıralama diyagramı üzerinde görüntüler. Şekil 6.1'deki bileşen diyagramında solda iki temel fonksiyona (bilgi toplama ve görüntüleme) göre bölünmüş asıl uygulama, sağda ise test amacıyla hazırlanmış ve Bölüm 6 senaryolarında kullanılan mikroservisler gösterilmiştir.



Şekil 6.1 XPLORA uygulaması bileşen diyagramı

Şekil 6.2'de XPLORA aracına ait daha detaylı UML 2.0 sınıf diyagramı bulunmaktadır. Uygulamanın iki temel bileşeninden biri olan XploraServiceLib, dış uygulamaların REST aracılığıyla gönderdiği olayları kabul etmekte, SequenceDiagramLib bileşeni ise bu olayları görselleştirmektedir. Aradaki ServiceMediator sınıfı iki bileşeni birbirinden soyutlayarak köprü görevi görür.



Şekil 6.2 XPLORA uygulaması sınıf diyagramı

6.1 XploraServiceLib Bileşeni

Görevi günlük olaylarını toplama fonksiyonunu yerine getirmektir. Diğer uygulamalar bu bileşen üzerinden olaylarını XPLORA aracına aktarırlar. Bileşenin

RESTsel metotlardan oluşan bir uygulama programlama arayüzü bulunmaktadır. Web hizmeti fonksiyonu için, servis yönelimli mimariler için yaygın olarak kullanılan bir çerçeve olan WCF (Windows Communication Foundation) kullanılmıştır.

6.2 SequenceDiagramLib Bileşeni

Sıralı diyagramların görselleştirilmesi amacıyla hazırlanmış bir Windows işletim sistemi kontrolü içermektedir (Egrilmez, 2020). Bileşen, Model-View-Presenter örüntüsüne göre tasarlanmıştır. Model kısmında veri yapıları, Görüntü (View) kısmında kontrolün görsel yönlerini, Sunum (Presenter) kısmı ikisi arasında iletişimi sağlayacak şekilde üçe bölünmüştür.

PlantUML (2020) kendi biçimleme dilinde hazırlanmış metin dosyalarından statik UML diyagram görüntüleri oluşturan bir kütüphanedir. Karşılaştırma imkanı sağlaması bakımından bu bölümdeki örnekler PlantUML tarafından sağlanan örneklerle uyumlu hazırlanmıştır.

Diyagram kontrolünün modelini oluşturan sınıflar şunlardır:

6.2.1 Sıralama (Sequence) Sınıfı

Sıralama, sıralamaya ait bütün bilginin saklandığı sınıftır. Katılımcılar arasındaki haberleşmenin, bu iletilerin zamanları ve hangi sırada gönderildikleri bilgisini de bulundurur.

API Kullanımı

```
sequence.Clear();
```

Sıralamanın içerdiği bütün bilgiyi siler. Sıralama ilk durumuna döner.

Dönüş nesnesi: (Yok)

Parametreler: (Yok)

```
sequence.Tick();
```

Sıralama diyagramlarının hedefi, bir sistemin ve üyelerinin zaman içindeki etkileşimini ve haberleşmesini görselleştirmektir. x eksenı sıralamanın üyelerini gösterirken, y eksenı zamanı gösterir. SequenceDiagramLib sıralama diyagramı kontrolünde, zaman tıkları ayırık değerler olarak ifade edilir. Her bir zaman tıki gerekleştiiğinde, sıralama bir sonraki zaman adımına ilerler. Zaman tıki sistemin kontrol tarafından durdurulduđu ve kullanıcıların müdahale edebileceđi noktalardır.

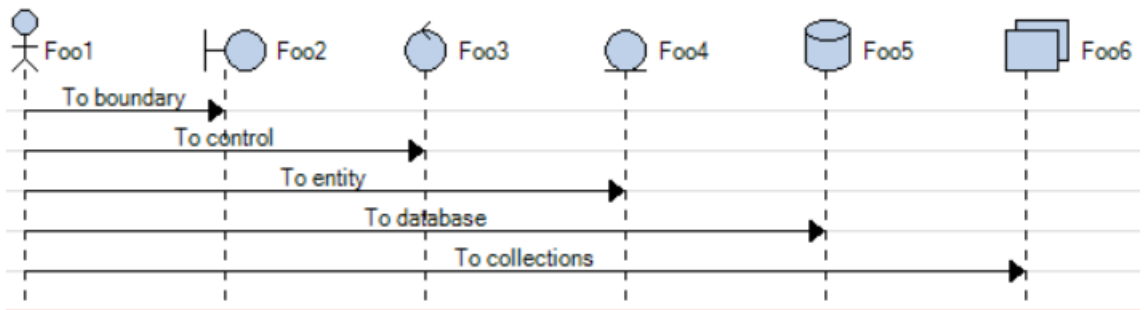
Dönüş nesnesi: (Yok)

Parametreler: (Yok)

6.2.2 Katılımcı (Participant) Sınıfı

Katılımcılar sıralamaya dahil olan temsilci ve nesnelerdir. Genelde diyagramın en üstünde yatay olarak yanyana dizilirler. Katılımcı normalde bir dikdörtgen ile temsil edilir ve adı dikdörtgenin içinde belirtilir. Adın alt çizgi ile yazılması, katılımcının bir sınıfı deđil, sınıftan türemiş bir nesneye ait olduğunu ifade eder. Katılımcı dikdörtgeni içine UML şablonları gibi ek bilgiler de eklenebilir.

Katılımcının yaşam çizgisi, altından başlayarak aşağı yönde uzanan düşey bir kesikli çizgi ile gösterilir. Katılımcının zaman içindeki yaşamı ve etkileşimini temsil eder. Şekil 6.3’de katılımcı sınıfına ait bir görselleştirme bulunmaktadır.



Şekil 6.3 Katılımcı sınıfının görselleştirilmesi

API Kullanımı

```
sequence.Participants.Create();
```

Yeni bir katılımcı oluşturur, sıralama içine yerleştirir ve bu katılımcı nesnesini döndürür. Sıralamada aynı isimde bir katılımcı zaten mevcutsa çağrı başarısız olur.

Dönüş nesnesi: **Participant**

Parametreler:

ad (string): Katılımcının adı.

altcizgi (bool): Katılımcı adının alt çizgili olarak gösterilip gösterilmeyeceği. UML diyagramlarında alt çizgi bir sınıfın belli bir nesnesini temsil eder.

renk (Color): Katılımcının arka plan rengi.

metinRengi (Color): Katılımcı adı rengi.

tip (EParticipantType): Katılımcı görüntü tipi.

Kutu (Box): Katılımcının hangi kutuya ait olacağı. Bu değer boş olabilir.

simdiOlustur (bool): Bu değer verilmesi, katılımcının içinde bulunulan zaman adımıyla oluştuğunu ve yaşamının başladığını vurgular. Değer verilmediğinde, katılımcı diyagramın en üst kısmında görüntülenir.

```
sequence.Participants.CreateOrGet();
```

Verilen isimde katılımcı mevcutsa, bu katılımcı nesnesini döndürür.

Verilen isimde katılımcı mevcut değilse yeni bir katılımcı oluşturur, sıralama içine yerleştirir ve bu katılımcı nesnesini döndürür.

Dönüş nesnesi: **Participant**

(Parametreler bir üstteki metotla aynı)

```
sequence.Participants[];
```

Verilen isme ait katılımcı nesnesini döndürür. Eğer sıralamada aynı isimde bir katılımcı yoksa çağrı başarısız olur.

Dönüş nesnesi: Participant

Parametreler:

name (string): Katılımcının adı.

```
participant.Destroy();
```

Katılımcıyı yok eder.

Dönüş nesnesi: (Yok)

Parametreler: (Yok)

6.2.3 İleti (Message) Sınıfı

İletiler bir katılımcıdan diğerine gönderilen bir bilgiyi, bir sinyalin gönderim ve alımını, bir işlemin başlangıç ve yürütümünü yada yapılan bir çağrıyı temsil edebilirler. İletiler eş zamanlı yada eş zamansız olabilirler. Şekil 6.4’de ileti sınıfına ait bir görselleştirme bulunmaktadır. Katılımcılar kendi kendilerine de ileti gönderebilir (Şekil 6.4’de: ❶).



Şekil 6.4 İleti sınıfının görselleştirilmesi

API Kullanımı

`sequence.Messages.Add()`

Kaynak ve hedef katılımcılar arasında bir ileti oluşturur.

Dönüş nesnesi: Message

Parametreler:

`ad (string)`: İletinin adı.

`kimden (Participant)`: İletiyi gönderen katılımcı.

`Kime (Participant)`: İletinin gönderildiği katılımcı.

`renk (Color)`: İletinin görselleştirildiği renk.

`arrowhead (EArrowHead)`: İleti okunun baş kısmının gösterim şekli.

`cizgiTipi (DashStyle)`: İleti çizgisinin çizim şekli.

`sequence.Messages.Add()`

Bir öz ileti oluşturur. Katılımcı kendi kendisine ileti gönderir.

Dönüş nesnesi: Message

Parametreler:

`ad (string)`: İletinin adı.

`kendi (Participant)`: İletinin sahibi katılımcı.

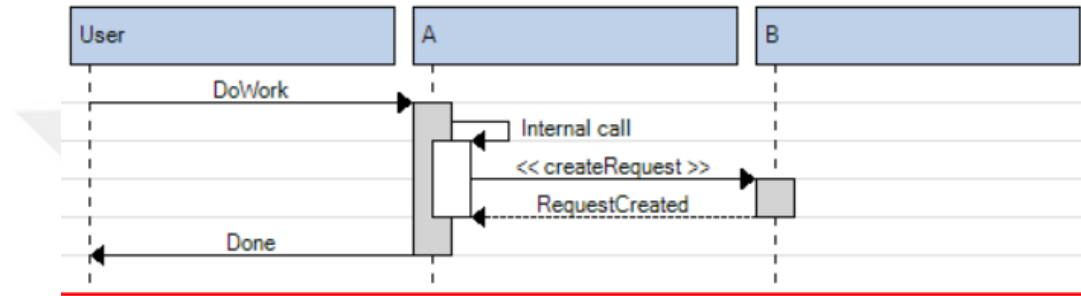
`renk (Color)`: İletinin görselleştirildiği renk.

`okUcu (EArrowHead)`: İleti okunun baş kısmının gösterim şekli.

`cizgiTipi (DashStyle)`: İleti çizgisinin çizim şekli.

6.2.4 Etkinleşme Kutusu (Activation Box) Sınıfı

Etkinleşmeler bir katılımcının işlem yürütmekte olduğu zaman aralığını temsil eder. Katılımcının meşgul olduğu yada bir cevabı bekleyerek geçirdiği bu zaman, yaşam çizgisinin üzerine yerleştirilmiş düşey bir dikdörtgen ile gösterilir. Dikdörtgenin üst ve alt kısımları, işlemin başlangıç ve bitiş zamanına denk gelir. Etkinleşmeler özyinelemeli olabilirler. Şekil 6.5’de etkinleşme sınıfına ait bir görselleştirme bulunmaktadır.



Şekil 6.5 Etkinleşme kutusu sınıfının görselleştirilmesi

API Kullanımı

`participant.Activate()`

Bir katılımcı için etkinleşme başlatır. Etkinleşme özyinelemeli olabilir.

Dönüş nesnesi: (Yok)

Parametreler:

ad (string): Kutunun adı.

renk (Color): Kutunun rengi.

`participant.Deactivate()`

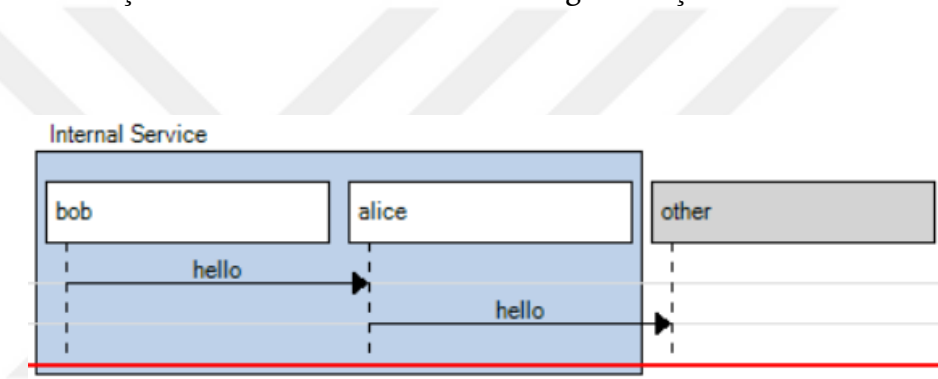
Katılımcıya ait mevcut etkinleşmeyi sonlandırır.

Dönüş nesnesi: (Yok)

Parametreler: (Yok)

6.2.5 Kutu (Box) Sınıfı

Kutular sıralama diyagramlarının daha anlaşılır bir şekilde düzenlenmesini sağlayan yardımcı sınıflardır. Birbiriyle ilişkili katılımcılar bir kutunun içinde gruplanabilir. Şekil 6.6'da kutu sınıfına ait bir görselleştirme bulunmaktadır.



Şekil 6.6 Kutu sınıfının görselleştirilmesi

API Kullanımı

`sequence.Boxes.Create()`

Yeni bir kutu oluşturur, sıralama içine yerleştirir ve bu katılımcı nesnesini döndürür. Sıralamada aynı isimde bir kutu zaten mevcutsa çağrı başarısız olur.

Dönüş nesnesi: Box

Parametreler:

ad (string): Kutunun adı.

renk (Color): Kutunun rengi.

`sequence.Boxes.CreateOrGet()`

Verilen isimde kutu mevcutsa, bu katılımcı nesnesini döndürür.

Verilen isimde kutu mevcut değilse yeni bir katılımcı oluşturur, sıralama içine yerleştirir ve bu katılımcı nesnesini döndürür.

Dönüş nesnesi: **Box**

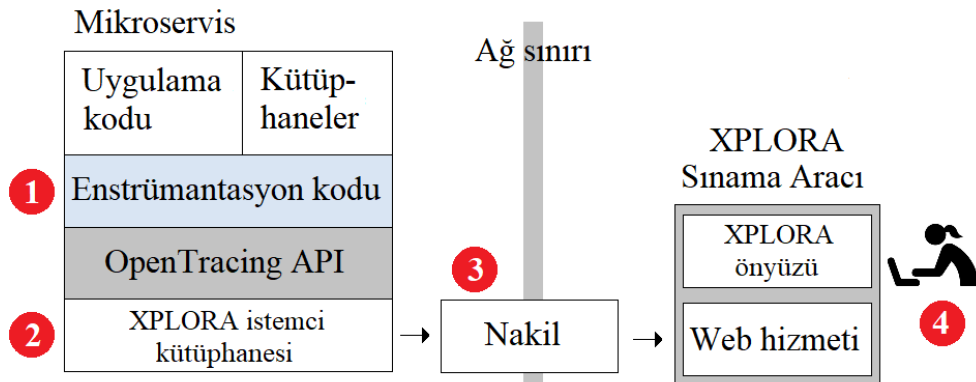
(Parametreler bir üstteki metotla aynı)



XPLORA ARACI KULLANIM SENARYOLARI

XPLORA, mikroservis mimarisi gibi dağıtık yapıdaki uygulamaların yürütümünü sıralama diyagramı tabanlı bir kullanıcı arabirimi üzerinden izlenebilmesi ve denetimini amaçlayan gerçek zamanlı bir yazılım sınaama aracıdır. Açık kaynak kodlu OpenTracing (OpenTracing, 2020) çerçevesini desteklemesi, aracın OpenTracing uyumlu enstrümantasyon yapılmış tüm uygulamalarda kullanılabilmesini mümkün kılar. Önceki bölümlerde tarif edilmiş geleneksel dağıtık izleyicilerden farkı, yakaladığı izleme olaylarını gerçek zamanda görsellenen duraksatma noktaları olarak kullanmasından gelir.

Şekil 7.1'deki diyagramda XPLORA uygulamasının ana bileşenleri gösterilmektedir. Uygulama yürütümü sırasında, enstrümantasyon kodunca oluşturulan iz günlük olayları, uygulamadan dağıtık izleyici istemci kütüphanesine geçer (Şekil 7.1'de: ❶). 'XPLORA İstemci Kütüphanesi' (Şekil 7.1'de: ❷), enstrümantasyon tarafından oluşturulan bu olayları RESTsel çağrılarını kabul eden bir web hizmeti olan 'XPLORA Sınama Aracı'na iletir (Şekil 7.1'de: ❸). Araç, kendisine ulaşan bu olayları bir sıralama diyagramı içine yerleştirir ve bu diyagramı sinayıcının interaktif olarak görüntülemesini mümkün kılar (Şekil 7.1'de: ❹).



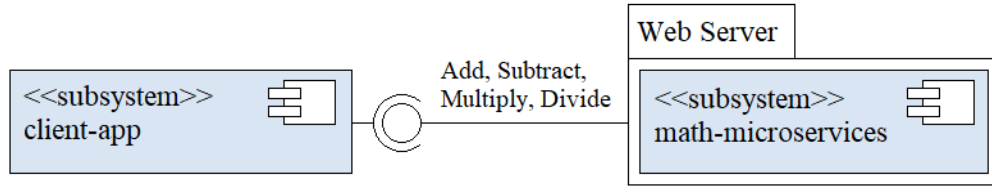
Şekil 7.1 Enstrümente edilmiş bir uygulama ve XPLORA aracına ait diyagram

Şekil 3.11'deki Uber Jaeger ve Şekil 6.1'deki XPLOA bileşen diyagramı karşılaştırıldığında, iki şeklin hem çok benzeyen, hem çok farklı kısımları olduğu görülür. Sınama altındaki mikroservis ve enstrümantasyon kodu tamamen aynı kalırken, fark istemci kütüphanelerinden kaynaklanır. Bunu mümkün kılan özellik, OpenTracing API katmanının getirdiği soyutlamadır.

Bu bölümde dört farklı senaryo ile XPLOA sınama aracı işlevsel tanıtımı yapılmıştır.

7.1 Senaryo 1 – Bir İstemci ve Mikroservis Arasında Etkileşim

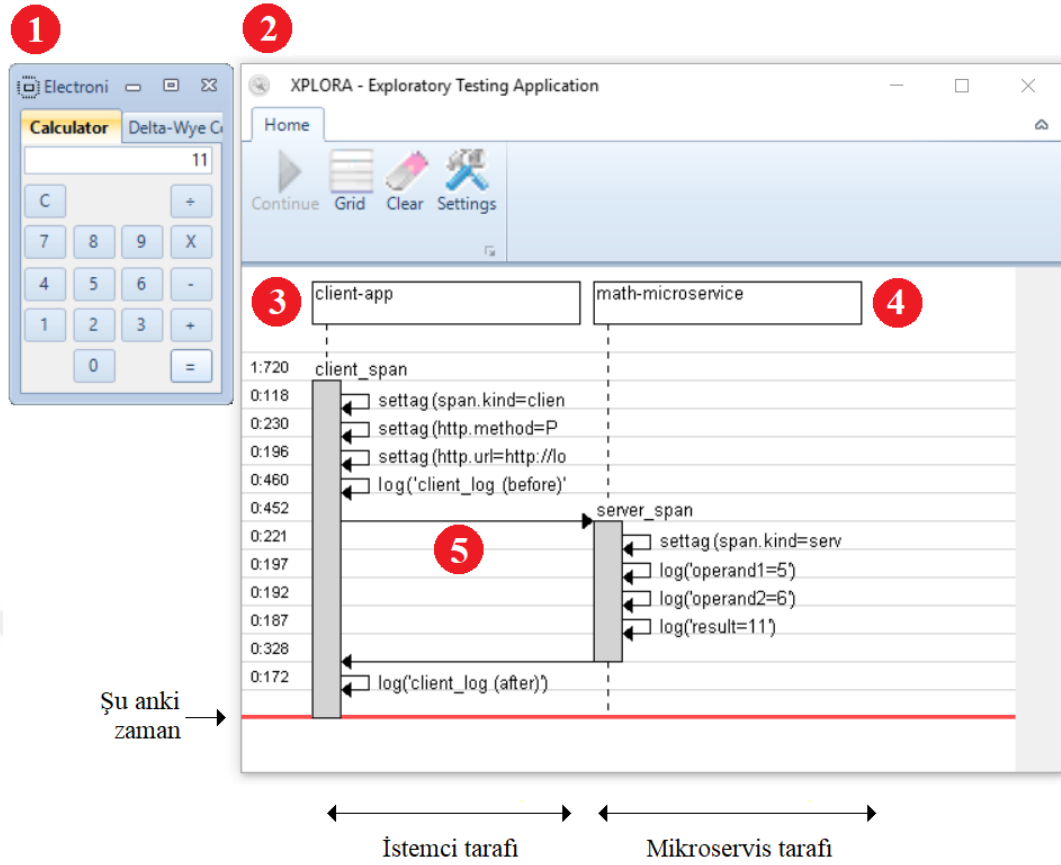
Tasarlanabilecek en basit dağıtık çözüm senaryosu iki adet yapı taşı içerebilir: bir sunucu bileşeni ve bu sunucuya istekte bulunan bir istemci bileşeni. Bu nedenle ilk senaryo bu mimari üzerine kurulmuştur.



Şekil 7.2 İstemci-mikroservis senaryosuna ait bileşen diyagramı

Şekil 7.2'de bu iki yapı taşına uyan bir bileşen diyagramı gösterilmiştir:

- Dört temel matematik işlemini yerine getirebilen math-microservice adlı mikroservis
- client-app ('Elektronik Tezgahı') adlı, arzu edilen hesapları yerine getirmek için bu mikroservisi kullanan ön yüz uygulaması. Sonraki senaryolarımızda bu uygulamaya yeni özellikler ekleyeceğiz.



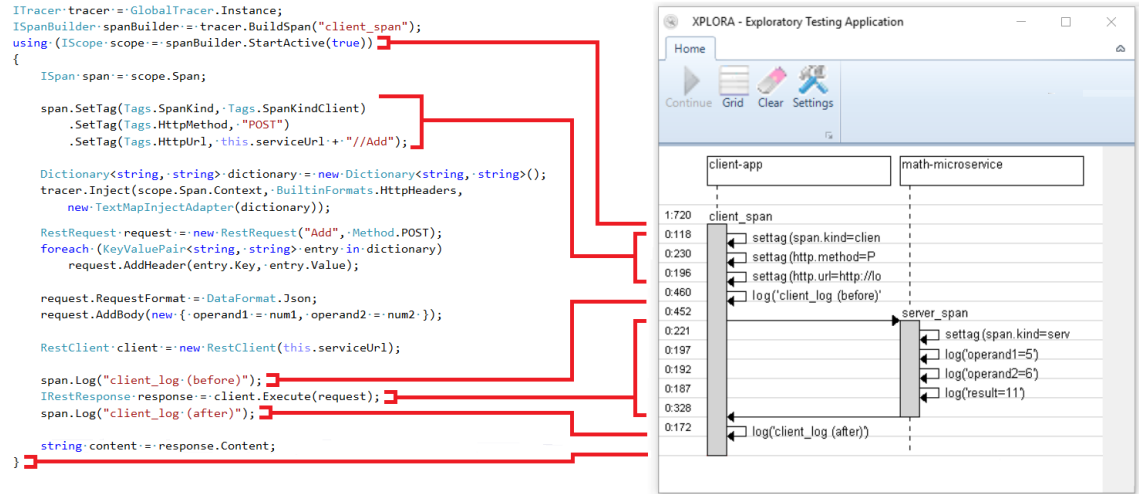
Şekil 7.3 İstemci-mikroservis senaryosunun XPLORA görselleştirmesi

Şekil 7.3'te bu senaryoya ait XPLORA görselleştirmesine yer verilmiştir: Solda hesap makinası kipindeki 'Elektronik Tezgahı' uygulaması bulunmaktadır (Şekil 7.3'te: ❶). Bu uygulama, math_microservice'in istemcisi durumundadır ve gerek duyduğu hesaplamalar için RESTsel HTTP metot çağrılar için bu mikroservisi kullanmaktadır. Sağ tarafta ise, XPLORA sınaa aracının ana ekranı görülmektedir (Şekil 7.3'te: ❷).

Sıralama diyagramında iki katılımcı görüntülenmektedir: client-app adlı sol katılımcı, 'Elektronik Tezgah' adlı uygulamaya denk gelir (Şekil 7.3'te: ❸). math-microservice adlı sağ katılımcı ise, dört işlemli matematik mikroservisimize denk gelir (Şekil 7.3'te: ❹). Yürütümün zaman eksenine göre ortalarında, 'Elektronik Tezgah' uygulaması, mikroservisi çağırılmaktadır (Şekil 7.3'te: ❺). Bu senaryodaki çağrı senkron olduğu için, client-app uygulaması çağrı sonuçlanana kadar beklemede kalır. Sınama altındaki uygulamadan ulaşan her günlük olayı, sıralama

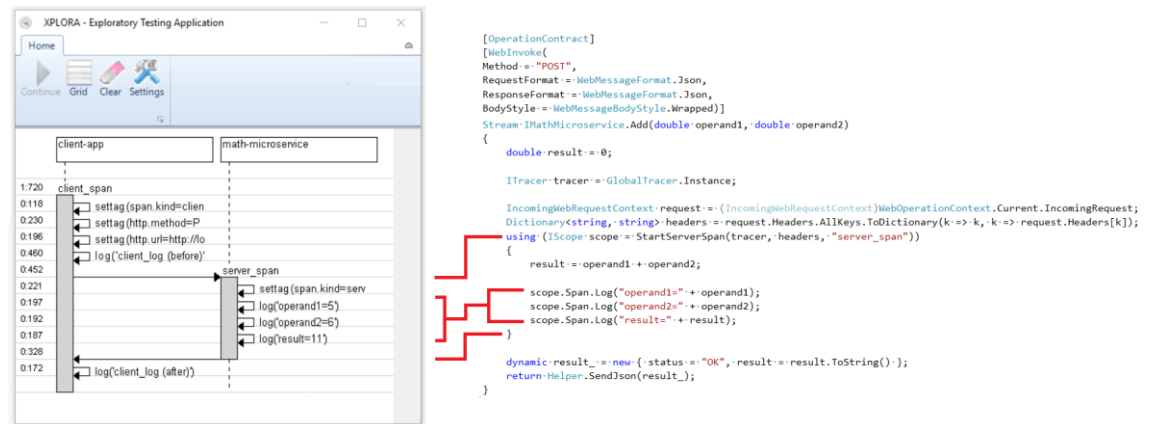
diyagramına eklenir. Sınamacının, bu olayları inceleme ve arzu ettiği şekilde denetim imkanı bulunur. Ekranın en altındaki kırmızı kalın çizgi, yürütümün o an bulunduğu zamanı ifade eder.

Şekil 7.4'de bu görselleştirmeyi sağlamak için gerekli enstrümantasyonun istemci tarafı gösterilmiştir.



Şekil 7.4 İstemci tarafında yapılan enstrümantasyon

Şekil 7.5'de ise bu görselleştirmeyi sağlamak için gerekli enstrümantasyonun sunucu tarafı gösterilmiştir.

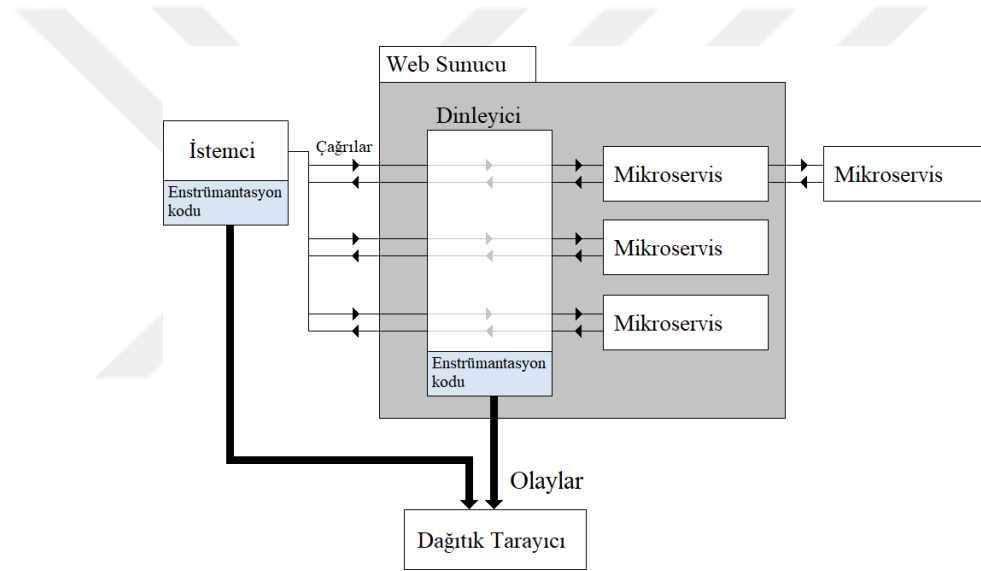


Şekil 7.5 Sunucu tarafından yapılan enstrümantasyon

7.2 Senaryo 2 – Dinleyici Enstrümantasyonu ile İz Oluşturma

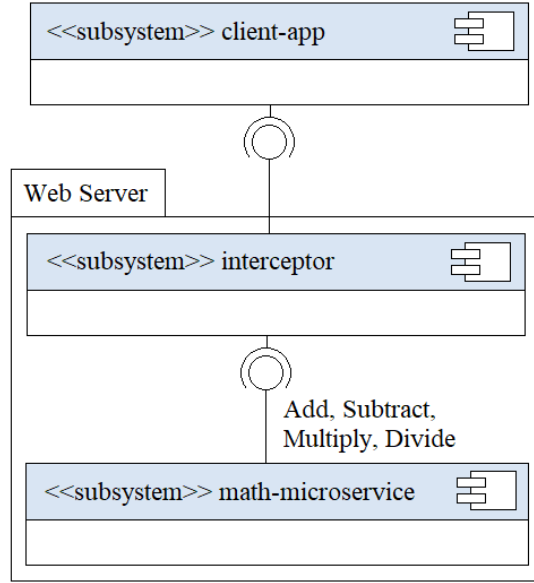
Bir yazılım bileşeninin koduna enstrümantasyon amacıyla kod eklenmesi her durumda tercih edilmeyebilir, hatta kodun değişikliklere kapalı olmasından dolayı mümkün de olmayabilir. Bu durumda, enstrümantasyonun bileşen tarafından kullanılan paylaşımlı kütüphaneler yada web sunucunun işlem akışı üzerine eklenen bir dinleyici üzerinden yerine getirilmesi bir çözüm olabilir.

Şekil 7.6'daki diyagram bu senaryoya aittir: İkinci senaryomuzda mikroservisler enstrümante edilemediği için, enstrümantasyon işlemi mikroservisleri ortak olarak kesen bir web sunucusu eklentisi üzerinde gerçekleştirilmiştir.



Şekil 7.6 Dinleyici enstrümantasyonu ile iz oluşturma

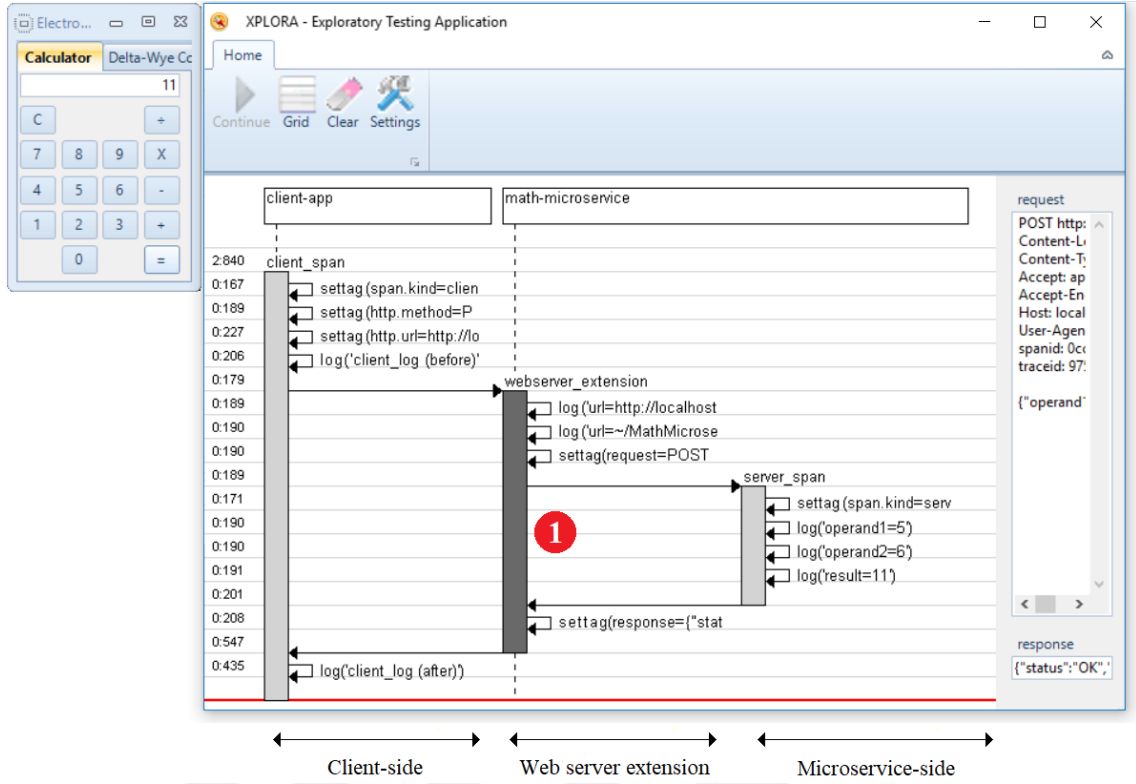
Şekil 7.7'deki bileşen diyagramında interceptor adı altında görüntülenen orta bileşen, bu eklentiye karşılık gelmektedir.



Şekil 7.7 Dinleyici enstrümantasyonu senaryosu bileşen diyagramı

Dışarıdan çalışmasına bakıldığında, bu senaryonun yürütümü aynı Bölüm 7.1'deki senaryo gibi görünmektedir. Ama dahili çalışmasında web hizmetine gelen bütün çağrılar enstrümante edilmiş interceptor bileşeni üzerinden geçer. Bu bileşen uygulama kodunda herhangi bir değişikliğe gerek olmadan, sadece web sunucusunun konfigürasyonu değiştirilerek etkinleştirilebilir yada devre dışı bırakılabilir.

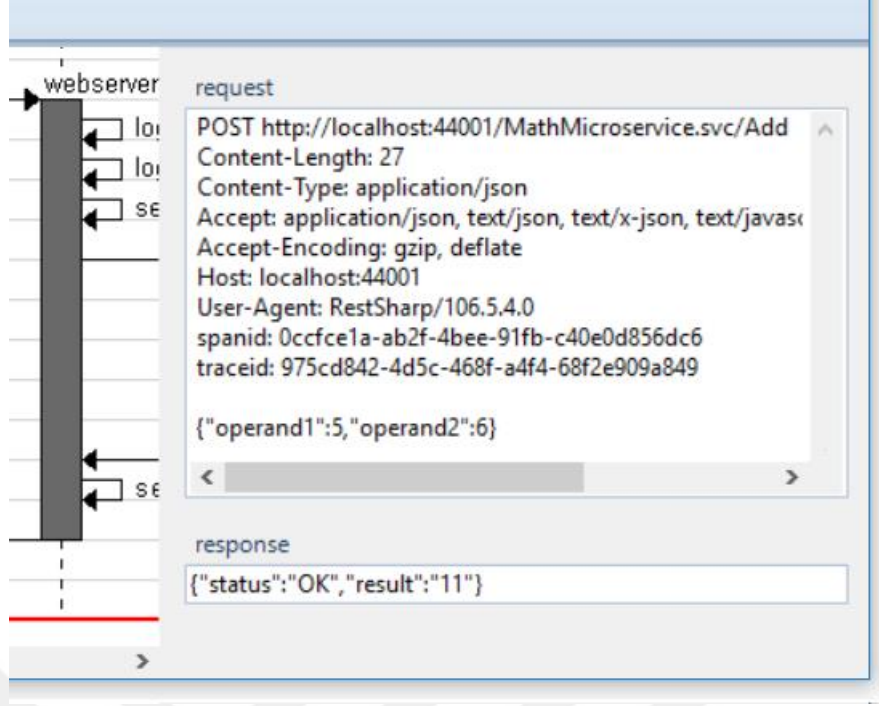
Şekil 7.8'deki ekran örneğinde, interceptor eklentimizi ifade eden katılımcının, yürütüm izi hiyerarşisi içinde, istemci uygulaması ve mikroservis arasında ortada ait olduğu yerde bulunduğu gözlemlenebilir (Şekil 7.8'de: ❶).



Şekil 7.8 Dinleyici enstrümantasyonu senaryosu XPLORA görselleştirmesi

Bu tip uygulama bağımsız, paylaşımlı kütüphaneler üzerinden gerçekleştirilen enstrümantasyon, sınam sürecinde bir çok avantaj sunar: XPLORA uygulaması, Şekil 7.9’da gösterildiği şekilde ekranın sağında bulunan ayrı bir uygulama panelinde, tüm metod çağrılarının hem işleme karşılık gelen ham HTTP istek ve yanıtlarını, hem de işlem sonucu oluşan JSON yüklerini gerçek zamanlı olarak görüntüleyebilir.

Diğer taraftan bu şekilde dolaylı olarak gerçekleştirilecek enstrümantasyonla oluşturulacak izin, elle yapılan enstrümantasyon kadar ince ayarlı olamayacağı gözden kaçırılmamalıdır.



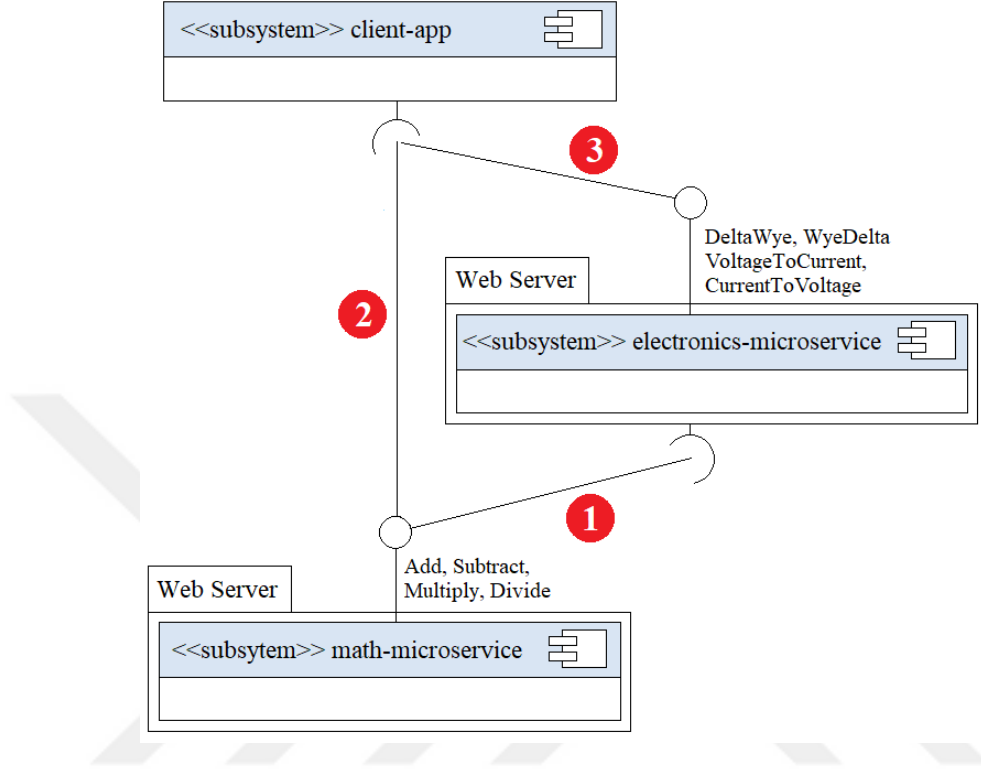
Şekil 7.9 İz etiketlerinin görselleştirilmesi

7.3 Senaryo 3 – İki Servis ve İstemci Arasındaki Etkileşim

Mikroservis mimarisinin karakteristik özelliklerinden biri bir işleme ait çağrı ağacının belirsiz adette farklı mikroservis arasında bölüşülebilmesi ve belirsiz derinlikte olabilmesidir. Bu bakımdan ilk iki senaryomuzun mikroservis mimarisinden çok klasik iki katmanlı istemci/sunucu mimarisine daha yakın olduğu söylenebilir. Bu eksikliği gidermek için üçüncü senaryomuzda uygulamamıza bir mikroservis daha ekleyerek çağrı derinliğini arttırılmıştır.

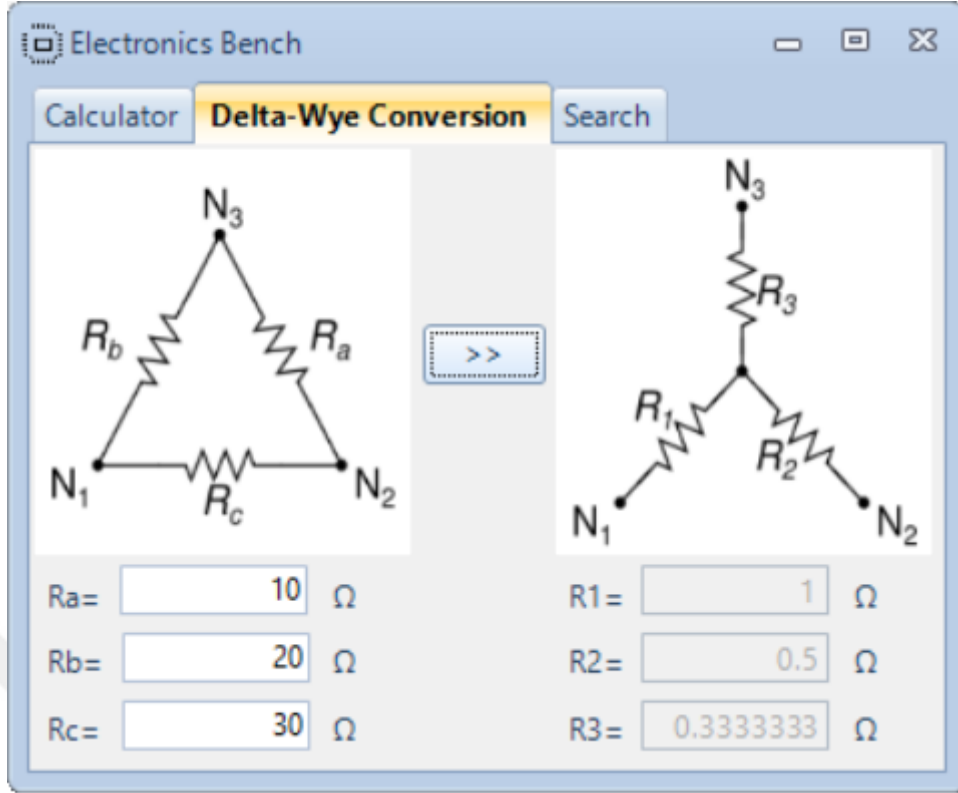
Şekil 7.10'da yeni senaryoya uyan bir bileşen diyagramı verilmiştir. Senaryoyla 'Elektronik Tezgah' uygulamamıza ek fonksiyonalitye eklenmiştir: Artık uygulama, temel aritmetik dört işlemini yerine getirmenin yanında, üçgen yıldız dönüşümü yada işlem amplifikatörü gerilim-akım dönüşümü hesaplamaları gibi orta seviyede elektronik hesaplamalar da yapabilmektedir. Bu elektronik hesaplamalar, electronics-microservice adındaki yeni mikroservisimizce yerine getirir. electronics-microservice ise math-microservice'in istemcisi durumundadır (Şekil 7.10'da: ❶), temel aritmetik işlemler için bu mikroservisi kullanır. client-app adlı

hesaplama ön yüz uygulamamız ise, hem math-microservice'in (Şekil 7.10'da: 2), hem electronics-microservice'in (Şekil 7.10'da: 3) istemcisi durumundadır.



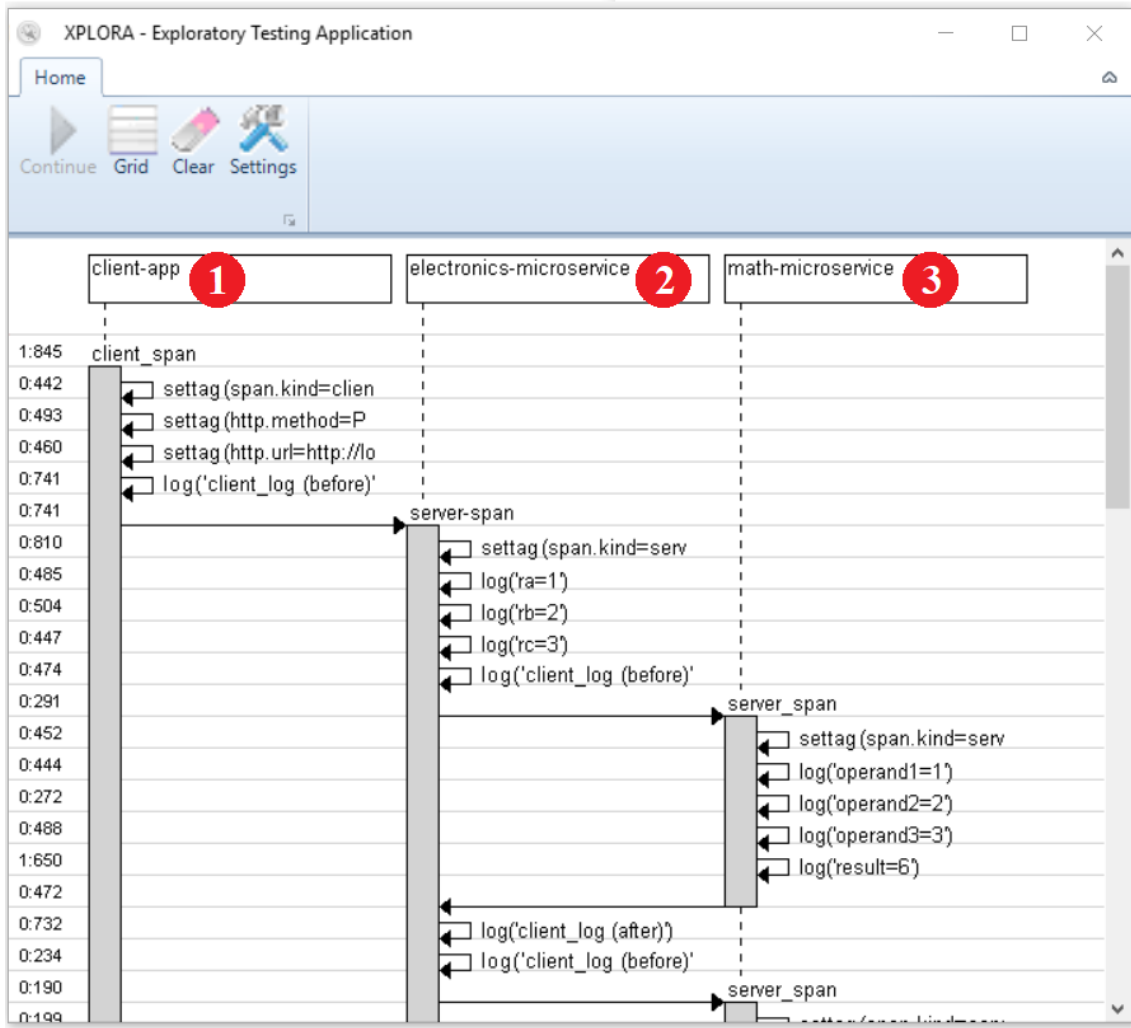
Şekil 7.10 Üç bileşen senaryosuna ait bileşen diyagramı

Şekil 7.11'deki ekran örneğinde, 'Elektronik Tezgahı' uygulaması bir üçgen-yıldız dönüşümü için kullanılırken görüntülenmiştir.



Şekil 7.11 Üç bileşen senaryosuna ait son kullanıcı ekranı

Bu işlemin XPLOA uygulaması görselleştirmesi, Şekil 7.12’de görüldüğü şekilde olmaktadır. Solda görüntülenen katılımcı, önyüz uygulamamız client-app (‘Elektronik Tezgahı’) bileşenine aittir (Şekil 7.12’de: ❶). Uygulama gerek duyduğu hesaplamalar için ortadaki katılımcı ile ifade edilen electronics-microservice’i çağırır (Şekil 7.12’de: ❷). electronics-microservice ise, temel seviye aritmetik işlemlerini içeren sağdaki katılımcı olan math-microservice’i çağırmaktadır (Şekil 7.12’de: ❸). Bir üçgen-yıldız dönüşümü için, 3 toplama, 3 çarpma ve 3 bölme işlemi gerektiği için, her bir hesaplamada electronics-microservice math-microservice’i 9 kere çağırmaya ihtiyaç duyar.



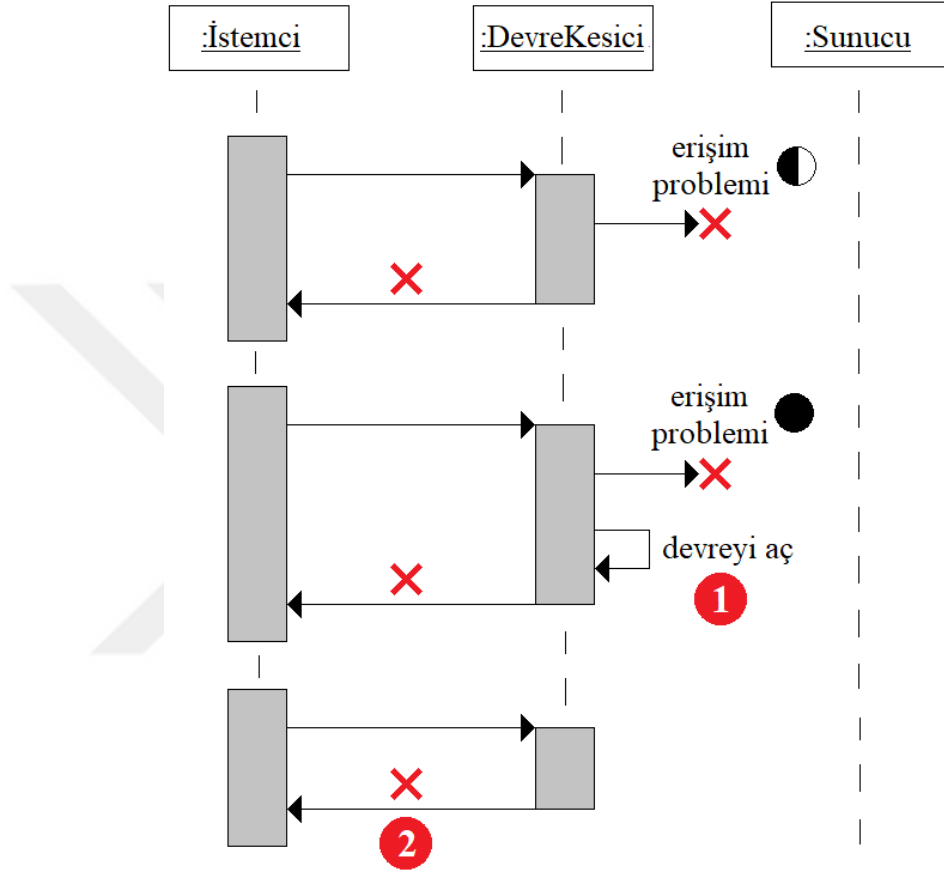
Şekil 7.12 Üç bileşen senaryosunun XPLORA görselleştirmesi

7.4 Senaryo 4 – Bağlantı Problemleri ve Devre Kesiciler

Mikroservis mimarisi bölünmüş yapısından dolayı yekpare bir çözüme göre ağ bağlantı ve gecikme problemlerinden çok daha fazla etkilenir.

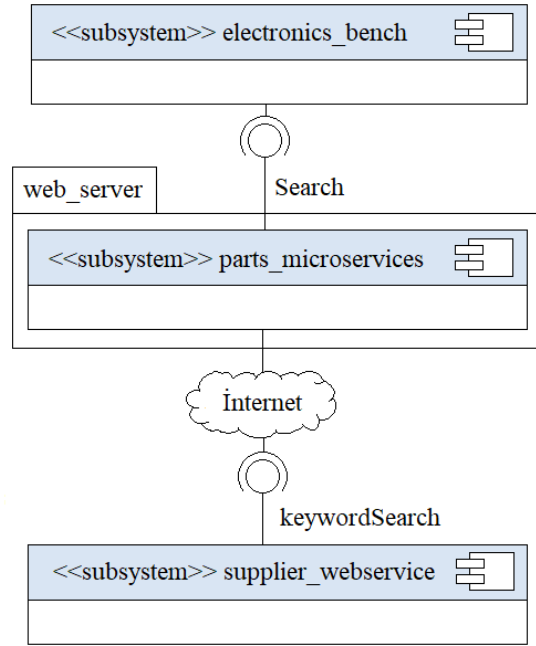
Şekil 7.13’de grafiksel olarak sunulan devre kesici, mikroservis mimarisindeki iletişim problemlerine karşı yaygın olarak kullanılan bir tasarım örüntüsüdür: Eğer bir işlem, yürütüm esnasında parametrik bir zaman aralığı içinde, parametrik olarak belirlenen adetten fazla başarısız olursa, bu işlemi kontrol eden devre kesicinin durumu kapalıdan açığa döner (Şekil 7.13’de: ❶). Açık kaldığı zaman aralığında, devre kesici ilgili işlemin yürütülmesini engeller (Şekil 7.13’de: ❷).

Üçüncü başka parametrik değer olan açık kalma zamanı dolduğunda, devre kesicinin durumu otomatik olarak tekrar kapalıya döner. Bu örüntünün mantığı, uygulama ve uygulamanın üzerinde çalıştığı sistem üzerindeki tahribatı arttırabileceği için, ardarda tekrarlanan başarısız denemelerin önüne geçmektir.

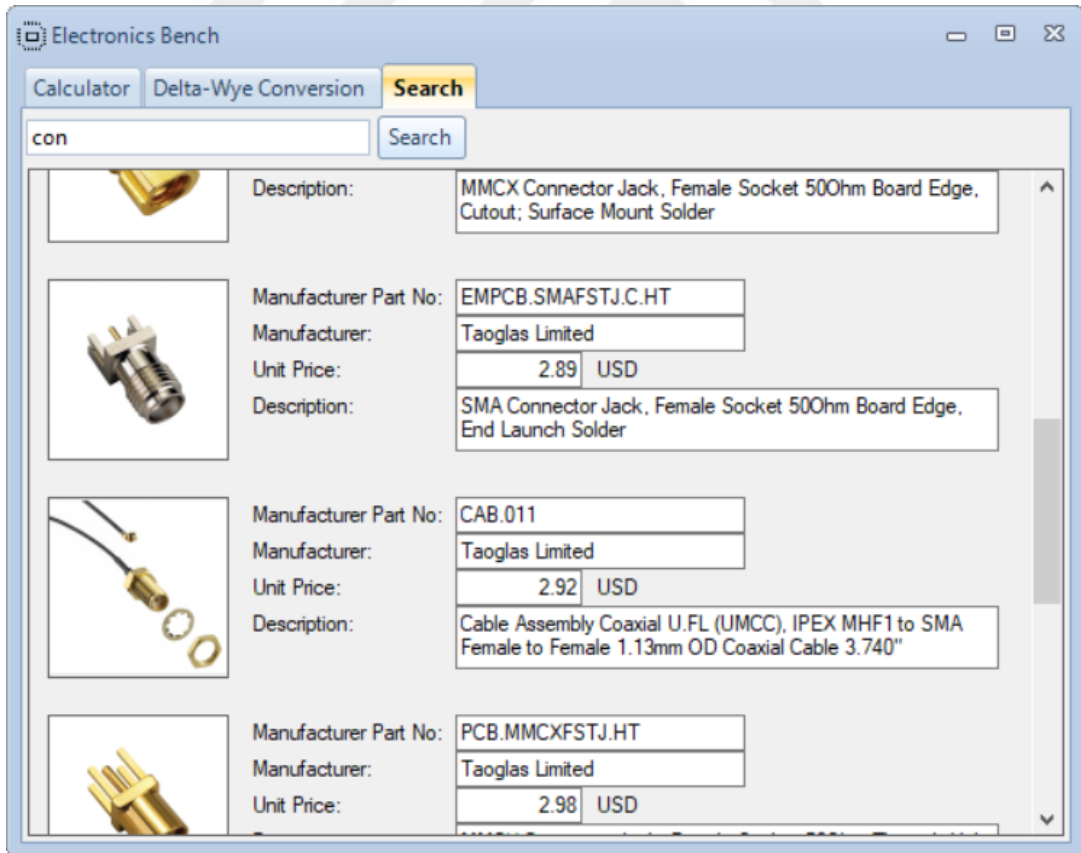


Şekil 7.13 Devre kesici

Şekil 7.14’de bileşen diyagramı verilmiş son senaryomuzda, ‘Elektronik Tezgahı’ uygulamamıza arama fonksiyonallitesi eklenmiştir: Kullanıcılarımız, standart bir arama motoruna benzer şekilde, ‘Elektronik Tezgahı’ önyüz istemci uygulamamıza sorgulamak istediği elektronik bileşenlere ait bir metin girebilirler. Uygulamamız da, bu metni dış bir web hizmetine geçit görevi olan ve sorgulayıp sonuç listesi oluşturmakla görevli olan parts-microservice adındaki mikroservise yönlendirir.



Şekil 7.14 Devre kesici senaryosuna ait bileşen diyagramı

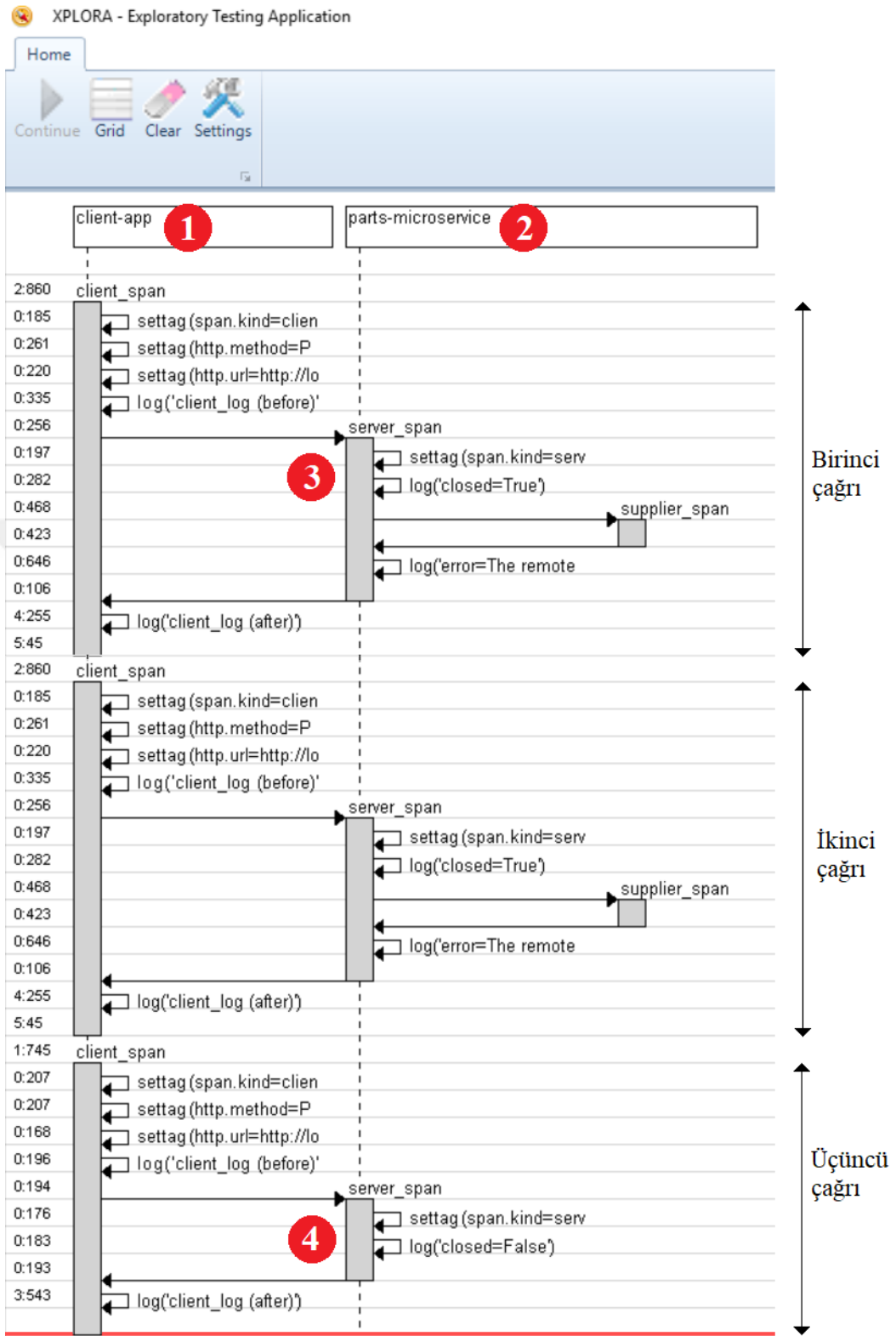


Şekil 7.15 Devre kesici senaryosuna ait önyüz ekranı

Şekil 7.15’de senaryoya ait önyüz ekranı bulunmaktadır. Devre kesiciye ait kod için Internette erişilebilen bir makaleden yararlanılmıştır (Desjardins, 2015). Kodu kendimiz yazma yerine hazır kod kullanmayı tercih etmemizin sebebi, hazır bir kodun enstrümente edilerek uygulama kodumuza entegre edilebileceğini göstermektir.

Şekil 7.16’da devre kesici senaryosunun XPLOA tarafından görselleştirilmesine yer verilmiştir. Soldaki client-app katılımcısı, ‘Elektronik Tezgahı’ istemci uygulamasını temsil etmektedir (Şekil 7.16’da: ❶). Sağdaki katılımcı ile temsil edilen parts-microservice’ini çağırır (Şekil 7.16’da: ❷). Digi-Key dış web hizmetine yapılan çağrı, en sağdaki etkinleşme kutusunda gösterilmektedir. Bu web servisin kendisi, bizim kontrolümüz dışında olduğu için, iç çalışması üzerinde herhangi bir enstrümantasyon yapılamaz, sadece yapılan çağrının kendisi görselleştirilebilir.

Sıralama diyagramında, Digi-Key web hizmeti erişilemez durumdayken uygulamamızın kendisini iki kere çağırmaya teşebbüs ettiği durum görüntülenmiştir: Devre kesicinin kapalı olduğu ilk çağrıda, parts-microservice’imiz gerçekten de çağrı denemesini gerçekleştirir, fakat başarısız olur (Şekil 7.16’da: ❸). İkinci başarısız denemeyle birlikte, devre kesici açık duruma geçer. Üçüncü çağrıda, açık durumunda kalmaya devam eden devre kesici, web hizmetine erişimi engelleyerek, hemen başarısız dönüş yapar (Şekil 7.16’da: ❹).



Şekil 7.16 Devre kesici senaryosunun XPLORA görselleştirmesi

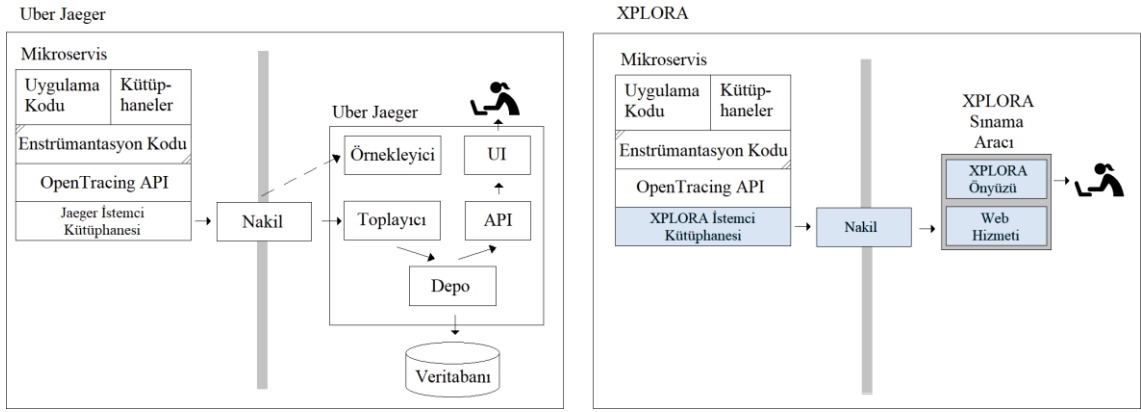
BENZER ÇALIŞMALAR İLE KARŞILAŞTIRMA

XPLORA sınaıa aracı dört farklı yazılım disiplini üzerine inşa edildiđi için, bu bölümde bu disiplinlere ait varolan uygulamalarla benzerlikleri ve farklılıkları ele alınmıştır.

8.1 XPLORA ve Dağıtık İzleyiciler

Ortak bir altyapıya dayanmakla beraber, XPLORA aracının dağıtık izleyicilerden en büyük farkı sıralama diyagramını uygulamanın işletimi sırasında, yani gerçek zamanlı olarak oluşturmasıdır. Böylece sınavıcı, uygulamaya ait değişkenlerin değerlerini anlık olarak görüntüleme olanağı elde eder. Bu durum popüler bir performans takip ve yazılım analiz çerçevesi olan Kieker Project’le (2017) karşılaştırılabilir: Kieker’ın uygulama yürütümüne yönelik (sıralama diyagramları da dahil olmak üzere) birçok UML diyagram tipi aracılığıyla görselleştirme desteğı bulunmaktadır. Ama vurgu; Graphviz, GNU PlotUtils gibi diyagram oluşturma araçlarını kullanarak, depolanmış iz verisinden geçmişe yönelik statik diyagramlar oluşturmaktır.

Tez çalışmasında açık yapılı OpenTracing dağıtık izleyici çerçevesi daha önce kullanılmayan bir şekilde kullanılmıştır. Şekil 8.1’de soldaki standart dağıtık izleyici uygulamasıyla, sağdaki XPLORA aracı karşılaştırılmıştır. Hazırlamış olduğumuz renkli gösterilen bileşenler aracılığıyla dinamik bir sınaıa aracına dönüşüm gerçekleştirilmiştir. Dağıtık izleyicilerde varolan depolama ve geçmişe yönelik analiz bileşenleri çıkarılarak yerine anlık görselleştirme ve analize yönelik bir yapı kurulmuştur.



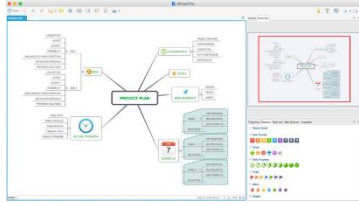
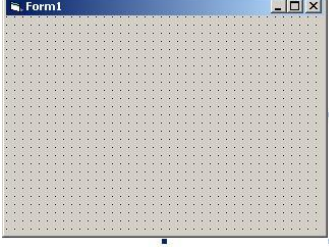
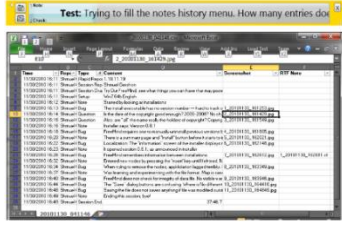
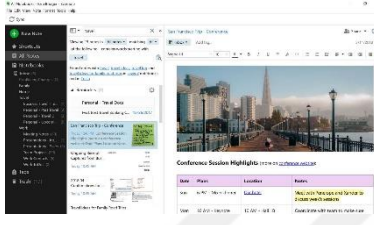
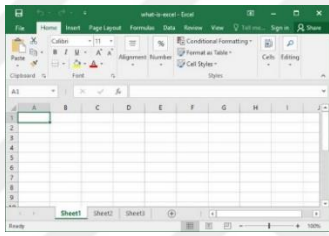
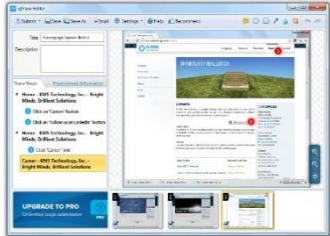
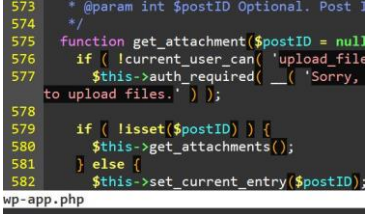
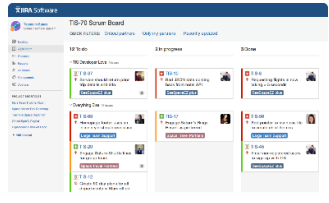
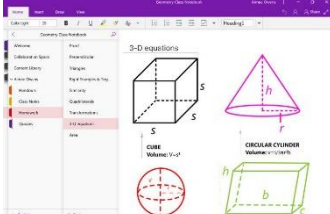
Şekil 8.1 Çalışmanın standart bir dağıtık tarayıcıyla karşılaştırılması

8.2 XPLORA ve Keşifsel Sınama Araçları

Finlandiya ve Estonya'daki sinayıcıların keşifsel sınamadan ne anladığını ve nasıl uyguladığını inceleyen bir araştırmada (Pfahl, Yin, Mantyla & Munch, 2014), keşifsel sınama tekniklerini uyguladıklarını söyleyen sinayıcıların %75'i herhangi bir keşifsel sınama aracı kullanmadıklarını bildirmişlerdir. Diğer %25 sinayıcının kullandıklarını bildirdikleri tüm araçlar Tablo 8.1'de listelenmiştir. Tabloda vurgulanmak istenen nokta bildirilen araçların aslında not alma, ekran kayıt, metin editörü, hesap tabloları gibi genel amaçlı uygulamalar olmasıdır.

Bu araçların haricinde Testpad (2020), PractiTest (2020) gibi sınama araçları, başlatma belgesi ve not alma gibi keşifsel sınamanın dokümantasyonuna yönelik eklentiler sağlamaktadır. Dinamik yürütüm görselleştirmeyi amaçlayan XPLORA benzeri keşifsel sınamaya yönelik bir araca rastlanmamıştır.

Tablo 8.1 Keşifsel sınav araçları

XMind Zihin haritası 	Sınavıcının kendi geliştirdiği uygulamalar 	Rapid Reporter Keşifsel not alma 
Evernote Not alma 	Excel Hesap tablosu 	qTrace Ekran yakalama 
Vim Editor Metin editörü 	Jira Problem takip 	OneNote Not alma 
Perlclip Test değeri oluşturma 	IETester Tarayıcı sürüm uyumu 	BB Flashback Ekran kayıt 

8.3 XPLOA ve Sıralama Diyagramı Kullanan Görselleştirme Araçları

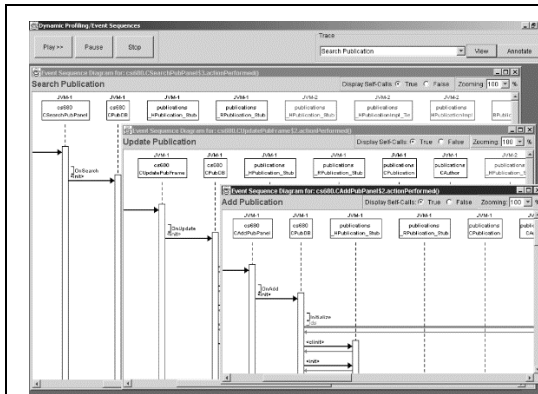
Tablo 8.2’de görselleştirme için sıralama diyagramı kullanan bazı yazılım araçlarının önyüzüne yer verilmiştir. Tablo 8.3’de ise bu özellikte bir yazılım görselleştirme aracının çözmesi gereken problemler ve bu problemlere yaklaşımları karşılaştırılmıştır.

Form çerçevesi (Souder vd., 2001), uygulamamızla benzerlik gösteren görselleştirme araçlarından bir tanesidir. Form, inceleme altındaki uygulamaya ait olayları Java Virtual Machine Profiler Interface (JVMPI) aracılığıyla yakalar. Bu sayede olaylar, enstrümantasyona yani uygulama kodu üzerinde değişikliğe gerek kalmadan yakalanmış olur. JVMPI’ya kayıt edilen olaylar gerçekleştikçe, JVMPI geri çağrılar aracılığıyla olayları Form çerçevesine bildirir.

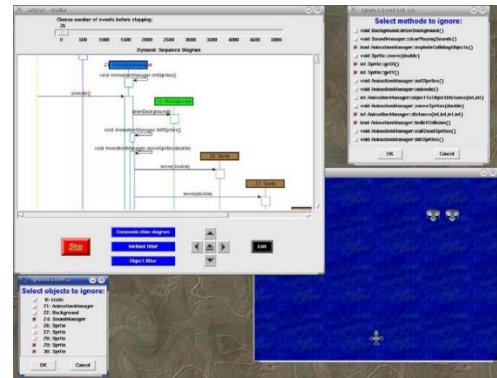
Form çerçevesinin salt Java tabanlı olması, mikroservislerin programlama dili bağımsız yapısı için yetersiz kalır. SD adlı görselleştirme aracının çevrimiçi ve çevrimdışı olmak üzere iki farklı çalışma şekli bulunmaktadır: Çevrimiçi şekil, uçtan uca yürütümü konsolde tek bir ekranda göstermek yerine her bir JVM için ayrı bir pencere oluşturur. OpenTracing’de bulunan bir açıklığı parametre değerleri ile ilişkilendirme yada XPLOA’nın parametre değerlerine dinamik ulaşma imkanlarına sahip değildir.

SPIDOR (Malloy & Power, 2005) ise uygulama olaylarını C++ dilinin boyuta yönelik programlama özelliklerini kullanarak yakalayarak bu olayları sınıf, sıralama ve haberleşme diyagramları aracılığı ile görselleştiren bir diğer yazılım aracıdır. Form çerçevesinde olduğu gibi enstrümantasyona ihtiyaç duymaz. Yalnızca C++ programlama dilini desteklemesi ve dağıtık uygulama desteği olmaması mikroservis mimarisiyle geliştirilmiş uygulamalar için kullanımının önündeki engellerdir.

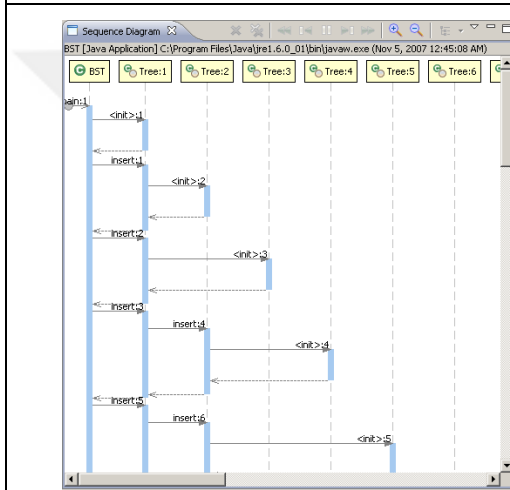
Tablo 8.2 Sıralama diyagramı kullanan görselleştirme araçları



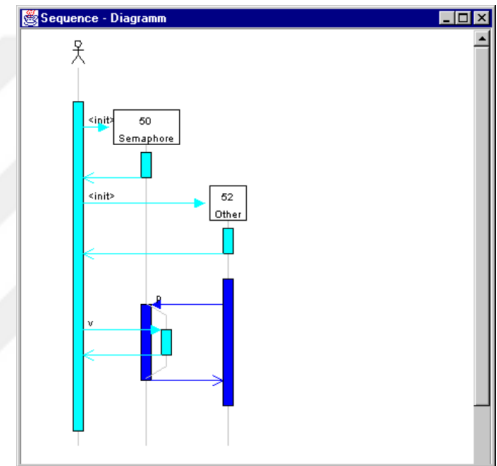
Form (Souder vd., 2001)



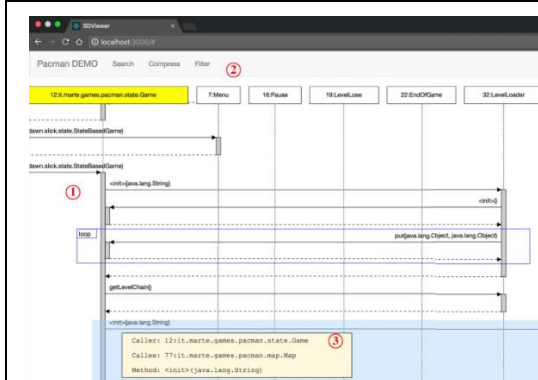
SPIDOR (Malloy & Power, 2005)



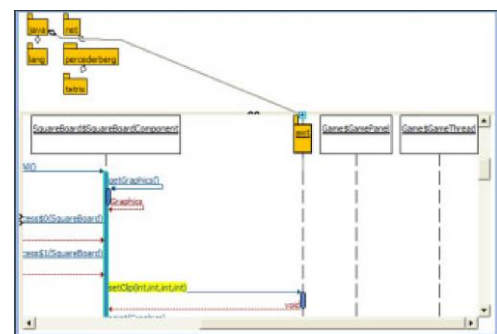
Jive (University of Buffalo, 2019)



JAVAVIS (Oechsle & Schmitt, 2002)



SDE Explorer (Lyu, Noda & Kobayashi, 2018)



OASIS (Bennett vd., 2008)

Tablo 8.3 Benzer görselleştirme çalışmaları ile karşılaştırma

Problem \ Araç	Form (Souder vd., 2001)	SPIDOR (Malloy & Power, 2005)	XPLORA
Enstrümantasyon Enstrümantasyon, uygulama geliştiricilerin yürütüm sırasında hangi olayların günlüklerinin tutulacağıının belirtebilmelerini sağlar.	İzleme için kod eklenmeye ihtiyaç duymaz. Olaylar Java Virtual Machine Profiler Interface (JVMPi) teknolojisi ile yakalanır. Form, kayıt olduğu olaylar gerçekleştiğinde yapılan JVMPi geri çağrılar ile yakalar.	İzleme için kod eklenmesine ihtiyaç uymaz. Olaylar, C++ tabanlı bir kütüphane olan AspectC++ ile yakalanır. SPIDOR, bu kütüphane aracılığıyla ilklendirici, imha edici ve metod çağrılarını yakalar.	Önceden kodun enstrümantasyonu yerine getirilmelidir. Enstrümantasyon OpenTracing tabanlı olduğu için, diğer OpenTracing destekli dağıtık izleyicilerle uyumludur.
Dağıtık İzleme ve Yayılım Dağıtık işlemler mikroservislerin temelini oluşturur: Bu gereksinim, dağıtık izleyici işlevi görecektir. Aracın, çağrı bağlam bilgisinin bileşenler arasında paylaşımına yönelik bir çözüm içermesini gerekli kılar.	Aracın dağıtık Java uygulamaları desteği vardır. Ama bu destek sadece Java Remote Invocation (RMI) içindir; mikroservisler tarafından yaygın olarak kullanılan RESTsel HTTP çağrılarını desteklemez.	Aracın tek süreç çalışması içinde desteği vardır. Dağıtık uygulama desteği bulunmamaktadır.	XPLORA, OpenTracing çerçevesinden HTTP üstbilgilerine dayanan yayılım mekanizmasını devralır.
Programlama Dilinden Bağımsızlık Uygulamanın her bir mikroservisi, hangi dil ihtiyacı daha iyi karşılıyorsa, o programlama dilinde yazılabilir. Bu nedenle dağıtık izleyici işlevi yerine getirecek araç, ayrı dillerdeki açıklıklardan oluşan bir çağrıya ait izi görüntüleyebilmelidir.	Araç, sadece Java ile geliştirilmiş uygulamaları destekler. JVMPi adlı bir Java kütüphanesine dayanmaktadır.	Araç, sadece C++ ile geliştirilmiş uygulamaları destekler. Bir C++ kütüphanesi olan AspectC++ kütüphanesini kullanmaktadır.	Araç, OpenTracing çerçevesi üzerine inşa edildiği için, kullanıcı topluluğu tarafından desteği sağlanmış bütün dilleri de destekliyor durumdadır. (Go, JavaScript, Java, Python, Ruby, PHP, Objective-C, C++, C#)
Desteklenen Diyagram Tipleri Ek diyagram tipi desteği, sıranan çözümün daha iyi anlaşılmasına yardımcı olur.	Sıralama diyagramları aracılığıyla görselleştirmeyi destekler.	Dinamik analiz için sıralama ve haberleşme diyagramlarını destekler. Statik analiz için sınıf diyagramını destekler.	Sıralama diyagramları aracılığıyla görselleştirmeyi destekler.

Bu çalışmada dağıtık izleyicilerin geçmişe yönelik analizi için oluşturulan iz olaylarını sıralama diyagramı elemanlarına dönüştürerek görselleştiren gerçek zamanlı bir keşifsel sınama aracı tanıtılmıştır.

Mikroservis uygulama mimarisinin bölünmüş yapısı bir taraftan uygulama geliştirmede avantajlar sağlarken diğer taraftan yürütümlerin sınanmasını, hata ayıklanmasını ve uçtan uçtan takibini zorlaştırır. XPLOA aracı farklı programlama dillerinde geliştirilmiş, farklı bilgisayarlara dağılmış bir yürütümün tek ve kolay anlaşılabilir bir ekranda görüntülenmesini mümkün kılar.

Keşifsel sınavıcıların yazılım sınaması ve analizi çabalarına yönelik birçok temel ihtiyaçları bulunur (Kaner & Bach, 2004, s. 16). Bunların en önemli üçü öğrenme (yazılım çözümünü nasıl öğrenebiliriz), görünürlük (yüzeyin altını nasıl görebiliriz) ve kontroldür (iç veri yapılarını nasıl değiştirebiliriz). XPLOA aracının sınama altındaki çözümü, yaygın bilinirliği olan bir diyagram tipi üzerinden izleyebilme ve sistemin iç durumunu gözlemleyebilme imkanı vererek bu üç soruna cevap verebilmektedir.

Tablo 9.1’de uygulamanın dayandığı dört yazılım mühendisliği disiplini listelenmiştir. İlk kolonda bu disiplinlerin ana fikirleri, yani varoluş amaçları özetlenmiştir. Daha sonra disiplinlerin bizce gelişmeye açık alanlarına değinilmiştir. Son kolonda aracın bu alanları ne şekilde tamamladığı ve yaptığı katkı anlatılmıştır.

Tablo 9.1 Çalışmanın ilgili yazılım mühendisliği alanlarına sağladığı katkı

	Ana Fikir	Gelişmeye Açık Alan	Katkımız
Mikroservisler	Uygulamayı tek bir bütün yerine temel yapıtaşlarına ayırmak; yazılım işbölümü, geliştirme, bakım alanlarında avantaj sağlar.	Uygulamanın bir çok otonom bileşene ayrılması; takibini ve sınanmasını zorlaştırır.	Dağıtık yazılım mimari uygulamalarının yürütümünün kolay takibi ve sınanmasına yönelik yeni bir yazılım aracı oluşturduk.
Dağıtık izleyiciler	Teknoloji bağımsız, modern, dağıtık bir uygulamanın takibi için klasik günlük mekanizması yetersiz kalır. Bunun için özel bir yazılım aracına ihtiyaç vardır.	Dağıtık izleyicilerle yapılan inceleme, analizin gerçek zamanlı değil, sonradan yapılmasına odaklanır.	Aracımız, sonradan analiz için tasarlanmış dağıtık izleme altyapısını, gerçek zamanlı analize imkan verecek şekilde zenginleştiriyor.
Yazılım görselleştirme	Yazılımın dinamik işletiminin anlaşılabilir bir şekilde görselleştirilebilmesi, analizcinin uygulamayı anlayabilmesi için için büyük fayda sağlar.	Varolan yazılım görselleştirme araçları, belli bir programlama dili yada platforma göre tasarlanmıştır. Günümüzün teknoloji bağımsız, modern dağıtık uygulamaları için yetersiz kalırlar.	Aracımız, OpenTracing adlı bir endüstri standardına dayandığı için, tüm yaygın yazılım platformlarını ve dillerini aynı anda destekliyor.
Keşif sınaması	Betikleştirilmiş sinama gibi otomasyona dayalı teknikler sinama için tek başına için yeterli değildir. Sinama otomasyon kadar bir düşünme aktivitesidir.	Keşif sınaması, varlığını betikleştirilmiş sinamaya bir tepki olarak devam ettirmektedir. Bu sinama tipi için yeterli somut araç desteği bulunmamaktadır.	Sinama altındaki sistemin anlaşılabilir şekilde görselleştirilebildiği, keşif sınaması ihtiyaçlarının göz önünde bulundurulduğu yeni bir yazılım aracı oluşturduk.

Tez çalışmasına eklenerek gelişmesini ve sağladığı katkıyı arttıracak bazı çalışma konusu önerileri şunlardır:

9.1 Sıralama Diyagramlarının Daha Etkin Kullanımı

Tez çalışmasında yakalanan kısıtlı sayıda iz olayının dinamik sıralama diyagramlarına dönüştürülmesi üzerine odaklanılmıştır. Farklı CASE araçlarının, sıralama diyagramlarını kullanırken yaptığı tercihleri karşılaştıran ve kullanışlı bir aracın uyması gereken kriterleri tespit etmeyi amaçlayan çalışmalar bulunmaktadır (Nikiforova, Putintsev & Ahilcenoka, 2016). Bu kriterleri inceleyerek ve çalışmamıza entegre ederek sınama aracımızın kullanılabilirliği arttırmayı amaçlıyoruz.

9.2 Sıralama Diyagramına Alternatif Diyagram Tipleri

Sıralama diyagramları, bir çözümün görselleştirilmesinde kullanılabilecek diyagram tiplerinden sadece birisidir. UML 2.0, sıralama diyagramlarını bir sistemi zaman içinde o sistem üzerinde oluşan değişiklikler dizisi olarak tarif eden davranışsal diyagramlar altında sınıflandırmıştır. Diğer davranışsal diyagramlar Gantt şemaları, akış grafikleri, çağrı grafikleri, çağrı bağlam grafikleri, nedensellik grafikleridir. Sıralama çizelgeleri (International Telecommunication Union, 2011) ve Petri ağları (Reisig, 2013) da nedensellik ilişkilerinin görselleştirilmesinde kullanılabilir. Bu diyagram tipleri Bölüm 4.2’de kısaca tanıtılmıştır.

İşlem akışına değil de sistemin yapısını, bileşenlerini ve bunların birbiriyle ilişkilerini görselleştirmeyi amaçlayan yapısal diyagramlar adlı ayrı bir UML 2.0 diyagram kategorisi de bulunmaktadır.

Takip eden çalışmalarda, bu alternatif diyagram tiplerinin dinamik görselleştirmesini sağlayarak uygulamamızı geliştirmeyi planlıyoruz. Alternatif diyagram tiplerinin eklenmesi, sinayıcının sınama altındaki sistemi görselleştirmesi ve anlamasına katkıda bulunacaktır.

9.3 Ek Senaryolar

Tez çalışmasında araç 4 senaryoyla tanıtılmıştır. Çalışmamızı şu ek senaryolarla zenginleştirmeyi planlıyoruz:

- Senkronizasyon ve paralel işleme içeren işlemlerin görselleştirilmesi
- API geçitleri gibi yaygın kullanımdaki mikroservis örüntülerinin görselleştirilmesi
- Orkestrasyon ve koreografi akış stillerinin görselleştirilmesi
- Kerberos, OAuth 2.0 gibi güvenlik protokollerinin görselleştirilmesi

9.4 Büyük İzler için Anlaşılabilir Görselleştirme

Yürütüm görselleştirmede karşılaşılan önemli bir problem, görselleştirilecek bilgi miktarı arttığında yada derinleştğinde, oluşturulan çıktıların anlamını koruyabilmesidir (Cornelissen vd., 2007). Milyonlarca olay eklendiğinde dahi diyagram sınavıcıya fayda sağlamaya devam etmelidir. Test aracına gerekli soyutlama ve filtreleme özellikleri ekleyerek sınavıcının büyük ölçekli izlerde de kaybolmadan çalışabilmesini, incelemek istediği veriye yoğunlaşabilmesini amaçlıyoruz.

9.5 Aracın Sağladığı Faydanın Nicel Ölçümü

Sınavıcının incelemekte olduğu çözümü daha iyi anlamasına yardım etmek, XPLOA sınaa aracının nihai amacını oluşturmaktadır. Anket benzeri tekniklerden faydalanılarak, gerçek yazılımcıların görüş ve önerilerini kayda almak yoluyla aracın sağladığı yararın nicel ve objektif olarak ölçmeyi planlıyoruz (Cornelissen, Zaidman & van Deursen, 2011).

- Apache Software Foundation. (2020a). Apache Kafka, Erişim adresi <https://kafka.apache.org>
- Apache Software Foundation. (2020b). Apache log4j 2, Erişim adresi <https://logging.apache.org/log4j/2.x/>
- Bach, J. (2002). Exploratory testing. In van Veenendaal, E. (Ed.), *The Testing Practitioner* (2nd ed., s. 261-274). Den Bosch, Hollanda: UTN Publishers.
- Bach, J. (2006). How to measure exploratory testing. Erişim adresi [http:// mail.quardev.com/content/whitepapers/how_measure_exploratory_testing.pdf](http://mail.quardev.com/content/whitepapers/how_measure_exploratory_testing.pdf)
- Barham, P., Isaacs, P., Mortier, R. & Narayanan, D. (2003). Magpie: Online modelling and performance-aware systems. Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems, 18-21 Mayıs 2003, Lihue (Kauai), Hawaii, ABD
- Bennett, C., Myers, D., Storey, M.D., German, D.M., Ouellet, D., Salois, M. & Charland, P. (2008). A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance*, 2008, s. 291-315.
- Chanda, A., Cox A.L. & Zwaenepoel, W. (2007). Whodunit: Transactional profiling for multi-tier applications. EuroSys '07 Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, 21-23 Mart 2007, Lizbon, Portekiz, s. 17-30
- Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A. & Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic Internet services. Proceedings International Conference on Dependable Systems and Networks, Washington DC, ABD, 2002, s. 595-604
- Chuvakin, A.A., Schmidt, K.J., Phillips, P. & Moulder, P. (2013). *Logging and Log Management*. Syngress, Waltham, Massachusetts, ABD

- Cornelissen, B., van Deursen, A., Moonen, L. & Zaidman, A. (2007). Visualizing testsuites to aid in software understanding. 11th European Conference on Software Maintenance and Reengineering (CSMR'07) (2007), s. 213-222
- Cornelissen, B., Zaidman, A. & van Deursen, A. (2011). A controlled experiment for program comprehension through trace visualization. IEEE Transactions on Software Engineering, 37 (3), s. 341-355
- Craig, R.D. & Stefan, P.J. (2002). *Systematic Software Testing*. Londra, Birleşik Krallık ve Norwood, Massachusetts, ABD: Artech House
- Desjardins, P. (2015). How to create a simple circuit breaker in C#. Erişim adresi <https://patrickdesjardins.com/blog/how-to-create-a-simple-circuit-breaker-in-c>
- Diehl, S. (2007). *Software visualization*. Berlin, Almanya: Springer-Verlag
- Digi-Key©. (2020). Digi-Key© API solutions. Erişim adresi <https://api-portal.digikey.com>
- Eğrilmez, M.B. (2020). A Dynamic Sequence Diagram Visualization Control. Erişim adresi <https://www.codeproject.com/Articles/5259009/A-Dynamic-Sequence-Diagram-Visualization-Control>
- Elastic. (2020). Elastic Kibana. Erişim adresi <https://www.elastic.co/products/kibana>
- Fonseca, R., Porter, G., Katz, R.H., Shenker, S. & Stoica, I. (2007). X-Trace: A pervasive network tracing framework. NSDI'07 Proceedings of the 4th USENIX conference on Networked systems design & implementation, 11-13 Nisan 2007, Cambridge, Massachusetts, ABD
- Fonseca, R., Dutta, P., Levis, P. & Stoica, I. (2008). Quanto: tracking energy in networked embedded systems. Proceedings of the 8th USENIX conference on operating systems design and implementation (OSDI'08), USENIX Association, 8-10 Aralık 2008, San Diego, Kaliforniya, CA, ABD, s. 323-338

- Fowler, M. (2014). Microservices. Erişim adresi <https://martinfowler.com/articles/microservices.html>
- Graham, D., Van Veenendaal, E., Evans, I. & Black, R., (2008). *Foundations of Software Testing: ISTQB Certification*. Boston, Massachusetts, ABD: Cengage Learning
- Gray, J. (2006). "A Conversation with Werner Vogels" ACM Queue, vol. 4, no. 4, s. 14–22.
- Hellerstein, J.L., Maccabee, M.M., Mills, W.N. & Turek, J.J. (1999). ETE: a customizable approach to measuring end-to-end response times and their components in distributed systems. Proceedings. 19th IEEE International Conference on Distributed Computing Systems, Austin, Teksas, ABD, 31 Mayıs - 4 Haziran 1999, s. 152-162
- Hoffman, M., Schnabel, E. & Stanley, K. (2016). *Microservices best practices for Java*. Raleigh, North Carolina, ABD: IBM Red Books
- International Telecommunication Union. (2011). Message sequence chart, Z.120 (02-2011). Erişim adresi <https://www.itu.int/rec/T-REC-Z.120>
- Itkonen, J. & Rautiainen, K. (2005). Exploratory testing: A multiple case study. 2005 International Symposium on Empirical Software Engineering (ISESE2005), 17-18 Kasım 2005, Noosa Heads, Queensland, Avustralya
- Jamshidi, P., Pahl, C., Mendonca, N.C., Lewis, J. & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead, IEEE Software, Mayıs/Haziran 2018, s. 24-35
- Kaner, C. (2008). A tutorial in exploratory testing. QAI QUEST Conference, Chicago, Illinois, ABD, 28 Nisan - 2 Mayıs 2008
- Kaner, C. & Bach, J. (2004). The nature of exploratory testing. Tampere, Finlandiya. Erişim adresi <http://www.kaner.com/pdfs/NatureOfExploratoryTest.pdf>

- Kaner, C., Falk, J. & Nguyen, H.Q. (1987). *Testing computer software*. Blue Ridge Summit, Pennsylvania, ABD: Tab Books
- Kempf, T., Karuri, K. & Gao, L. (2008). Software Instrumentation. In Wah, B. (Ed.), *Encyclopedia of Computer Science and Engineering*. Hoboken, New Jersey, ABD: John Wiley & Sons.
- Kieker Project. (2017). Kieker 1.13 user guide. Erişim adresi <http://kieker-monitoring.net/documentation/>
- Lee, H.B. & Zorn, B.G. (1997). BIT: A tool for instrumenting Java bytecodes. Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, Aralık 1997, s. 73-82
- Lyu, K., Noda, K., & Kobayashi, T. (2018). SDexplorer: a generic toolkit for smoothly exploring massive-scale sequence diagram. 2018 ACM/IEEE 26th International Conference on Program Comprehension, Gothenburg, İsveç, Mayıs, 2018, s. 380-384
- Malloy, A.M. & Power, J.F. (2005). Exploiting UML dynamic object modeling for the visualization of C++ programs. Proceedings SoftVis '05 - ACM Symposium on Software Visualization, s. 105-114
- Nikiforova, O., Putintsev, S. & Ahilcenoka, D. (2016). Analysis of sequence diagram layout in advanced UML modelling tools. Applied Computer Systems, De Gruyter Open 2016, vol. 19, s. 37-43
- Object Management Group. (2017). Unified Modeling Language Specification Version 2.5.1. Erişim adresi <https://www.omg.org/spec/UML/>
- Oechsle, R. & Schmitt, T. (2002). JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). The International Dagstuhl Seminar on Software Visualization, Mayıs 2001, Dagstuhl Kalesi, Almanya, s. 176-190
- OpenCensus. (2020). OpenCensus, Erişim adresi <https://opencensus.io>
- OpenTelemetry. (2020). OpenTelemetry, Erişim adresi <https://opentelemetry.io>

- OpenTracing. (2020). OpenTracing, Erişim adresi <https://opentracing.io>
- Our Sky Cartoons. (2018). Scripted testing vs. exploratory testing, Erişim adresi <https://www.facebook.com/hashtag/ourskycartoons>
- Parker, A., Spoonhower, D., Mace, J. & Isaacs, R. (2020). *Distributed tracing in practice*. Sebastopol, Kaliforniya, ABD: O'Reilly
- Pfahl, D., Yin, H., Mantyla, M.V., & Munch, J. (2014). How is exploratory testing used? A state-of-the-practice survey. Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '14). ACM, New York, NY, ABD
- PlantUML. (2020). PlantUML Sequence Diagrams. Erişim adresi <https://plantuml.com/sequence-diagram>
- PractiTest. (2020). Practitest. Erişim adresi <https://www.practitest.com/qa-learningcenter/exploratory-testing-qa-coverage/>
- Reisig, W., (2013). *Understanding Petri nets: Modeling techniques, analysis methods, case studies*. Berlin, Almanya: Springer-Verlag
- Richardson, C. (2019). *Microservices patterns*. Shelter Adası, New York, New York, ABD: Manning
- Sambasivan, R.R., Fonseca, R., Shafer, I. & Ganger, G.R. (2014). So, you want to trace your distributed system? Key design insights from years of practical experience. Technical Report, CMU-PDL-14, 2014
- Shkuro, Y. (2019). *Mastering Distributed Tracing*. Birmingham, Birleşik Krallık: Packt Publishing
- Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S. & Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report. Google, Inc., 2010. Erişim adresi <https://research.google.com/archive/papers/dapper-2010-1.pdf>
- Stori, D. (2017). The monolith retirement, Erişim adresi <https://dzone.com/articles/the-monolith-retirement-comic>

- Souder, T.S., Mancoridis, S. & Salah, M., (2001). Form: A framework for creating views of program executions. IEEE Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01), Floransa, İtalya, Kasım, 2001, s. 612-620
- Testpad. (2020). Testpad. Erişim adresi <https://ontestpad.com/features>
- Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J. & Ganger, G.R. (2006). Stardust: Tracking activity in a distributed storage system. Proceedings of the joint international conference on measurement and modeling of computer systems (SIGMETRICS '06/Performance '06). ACM, New York, NY, ABD, s. 3-14
- Twitter Zipkin. (2020). Zipkin Distributed Tracer. Erişim adresi <https://zipkin.io/pages/architecture.html>
- Uber Jaeger. (2020a). Uber Jaeger. Erişim adresi <https://www.jaegertracing.io>
- Uber Jaeger. (2020b). Uber Jaeger Sampling. Erişim adresi <https://www.jaegertracing.io/docs/1.13/sampling/>
- Unhelkar, B. (2018) *Software engineering with UML*. Boca Raton, Florida, ABD: CRC Press
- University of Buffalo. (2019). JIVE: Java interactive visualization environment. Buffalo, New York, ABD. Erişim adresi <https://cse.buffalo.edu/jive/>
- Whittaker, J.A. (2009). *Exploratory software testing*. Boston, Massachusetts, ABD: Addison-Wesley
- World Wide Web Consortium. (2018). Trace Context, W3C Working Draft. Erişim adresi <https://www.w3.org/TR/trace-context>

UBER JAEGER DAĞITIK İZLEYİCİ UYGULAMASI

XPLORA aracı dağıtık izleyici uygulamaları ile ortak özellikler içerir. Karşılaştırma imkanı oluşturması için bu bölümde yaygın olarak kullanılan Uber Jaeger dağıtık izleyici uygulaması tanıtılmıştır.

Uber Jaeger (2020), Uber şirketi tarafından ilk beta sürümü Temmuz 2017’de, 1.0 sürümü 6 Aralık 2017’de yayınlanmış bir dağıtık izleyici uygulamasıdır. Google Dapper (Sigelman vd., 2010) ve Twitter Zipkin (2020) gibi daha eski dağıtık izleyicilerden en büyük farkı, OpenTracing (2020) adlı açık çerçeveye dayanmasıdır.

A.1 Kurulum

Uygulamanın farklı işletim sistemleri (macOS, Linux, Windows) için oluşturulmuş kurulum paketleri, <https://www.jaegertracing.io/download/> adresinde bulunmaktadır (Şekil A.1). Bu paketlerden all-in-one.exe adlı exe, uygulamanın minimum yapılandırmayla çabuk test edilebilirliğini amaçlamaktadır.

```
D:\jaeger-1.7.0-windows-amd64>dir
Directory of D:\jaeger-1.7.0-windows-amd64

11/21/2018  02:26 PM    <DIR>          .
11/21/2018  02:26 PM    <DIR>          ..
09/19/2018  02:18 PM         15,413,248  example-hotrod.exe
09/19/2018  02:18 PM         22,180,864  jaeger-agent.exe
09/19/2018  02:18 PM         42,249,216  jaeger-all-in-one.exe
09/19/2018  02:18 PM         33,280,512  jaeger-collector.exe
09/19/2018  02:18 PM         36,628,480  jaeger-query.exe
              5 File(s)      149,752,320 bytes
              2 Dir(s)      494,036,426,752 bytes free

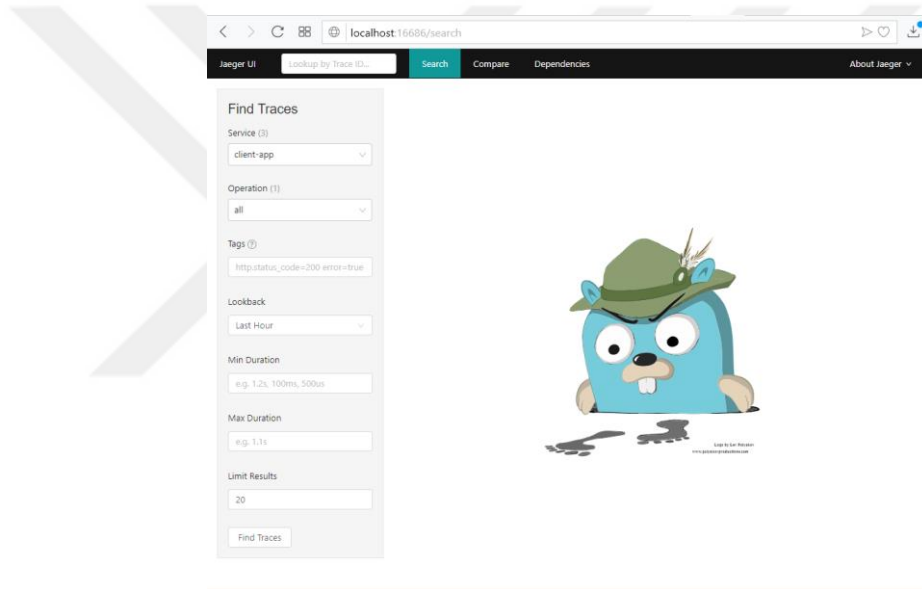
D:\jaeger-1.7.0-windows-amd64>jaeger-all-in-one.exe
```

Şekil A.1 Uber Jaeger kurulum paketi içeriği

all-in-one.exe'sinin alıřtırılmasıyla beraber, Uber Jaeger uygulamasının gerekli tm hafif bileřenleri aktif hale gelir.

A.2 Arayz

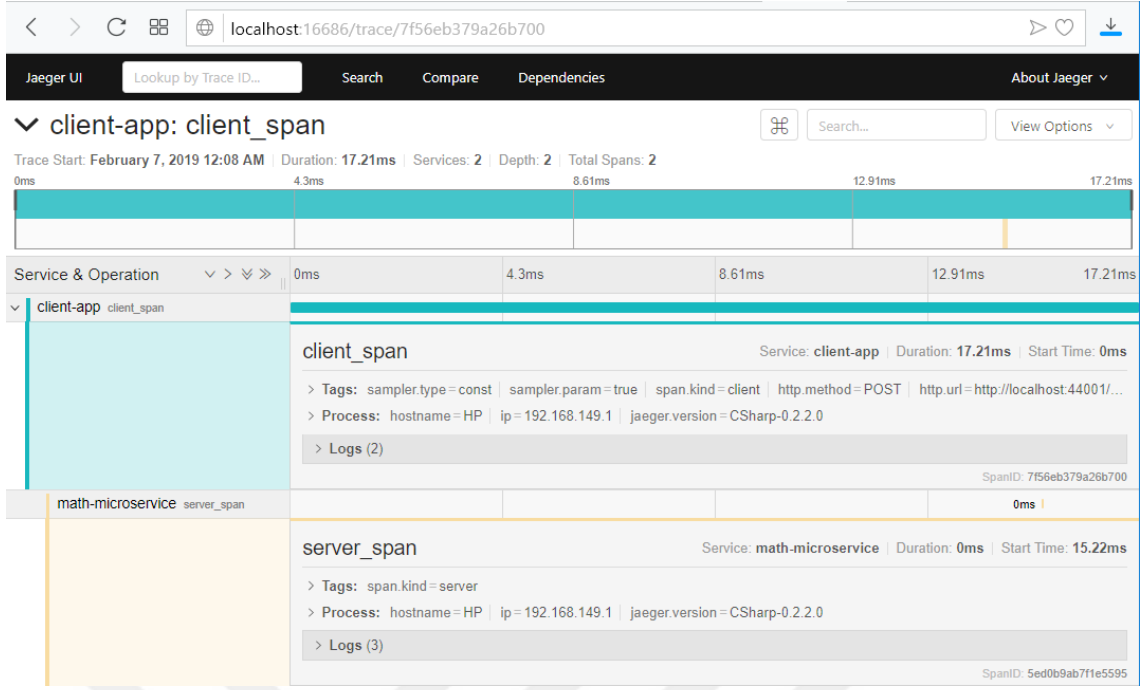
Karřılařtırma, Jaeger'in 1 numaralı basit matematik senaryomuza karřılık gelen durumu ile yapılmıřtır. 16686 port'undan řekil A.2'de gsterilen Jaeger'in ana sorgulama ekranına ulařılabilir.



řekil A.2 Uber Jaeger ana sorgulama ekranı

Bu ekrandan incelenmek istenen iřleme ait sorgu deęerleri girilerek, 'Find Traces' (İzleri bul) dęmesi ile arama sonucuna ulařılır. Arama sonucu ekranında, o ize ait btn iz aęacı, bir Gantt řeması halinde grntlenir.

Aıklıęa ait btn ek bilgilere (alt aıklıklar, gnlk olayları, etiket koleksiyonu) eriřilebilir. řekil A.3'de izlerin detay bilgilerinin grnts sunulmuřtur.



Şekil A.3 Uber Jaeger iz detay ekranı

Şekil A.3’de görüntüsü sunulan iz, Bölüm 7.1’deki XPLOA kullanım senaryosuna denk gelmektedir.

Tez çalışması dağıtık izleyiciler ve mikroservisler gibi henüz standartlaşma aşamasında olan disiplinlere dayandığı için, birçok terimin Türkçe sözlükler ve metinlerde kabul görmüş karşılıklarına rastlanılamamıştır. Bu terimler için aşağıda listelenen Türkçe-İngilizce terim eşleştirmesi yapılmıştır:

açıklık: span

alan odaklı tasarım: domain driven design

aktarım: transport

alım: reception

alt: child

API geçidi: API gateway

ayrıntılılandırılmış: fine-grained

ayrıştırmak: decompose

bağlam: context

bağlantı: connectivity

bakım: maintenance

barındırıcı orkestrasyonu: container orchestration

barındırıcılaştırma: containerization

başlatma belgesi: charter

betik: script

betikleştirilmiş sinama: scripted test

betimlemek: describe

bileşen: component

birimlere bölünmüş: modularized

bilgisayar ağı gecikmesi: network latency

boyut yönelik programlama: aspect oriented programming

böcek ilacı paradoksu: pesticide paradox

çağrı ağacı: call tree

çerçeve: framework

çok işlemli: multiprocessing

dağıtık: distributed

dağıtık izleyici: distributed tracer

denetim evirme: inversion of control

devreye alma: deployment

dinleyici: interceptor

dinlemek: intercept

doğaçlama: ad hoc

döngüsüz: acyclic

duraksatma noktası: breakpoint

durum çözümlemesi: what-if scenario

eklenti: plugin

enstrümantasyon: instrumentation¹

eş zamanlama: synchronize

¹ Enstrümantasyon kelimesi biyomedikal ve aviyonik gibi diğer bilim alanlarında “ölçüm” kelimesine denk gelse de, yazılım geliştirme alanında “performans ölçmek, hataları tespit etmek veya iz bilgisi oluşturmak için uygulamaya özel amaçlı kod eklenmesi” olarak özel bir kullanımı ifade etmektedir (Kempf, vd., 2009). Karşılığı olacak bir kelimeye rastlanılamadığı için Türkçeleştirilmemiştir.

eş zamanlı: synchronous

etkileşim: interaction

etkinleşme kutusu: activation box

etmen: agent

Gantt şeması: Gantt chart

geçmişe yönelik: retrospective

geri çağırma: callback

gevşek bağlı: loosely-coupled

görselleştirme: visualization

günlük oluşturma: logging

hata: bug

hata dayanıklı: fault tolerant

hata içitme: fault injection

hesap verilebilirlik: accountability

hizmet: service

hizmet ağı: service mesh

hizmet tabanlı mimari: service oriented architecture

icitme: injection

ileti aracısı: message broker

ileti kuyruğu: message queue

iklendirici: constructor

ince ayarlı: fine-tuned

iş parçacığı: thread

iş parçacığı yerel hafızası: thread local storage

işlem: operation
işlem: process
iz: trace
izleyici: tracer
kaos mühendisliği: chaos engineering
katılımcı: participant
kayıt olmak: register
kesintisiz tümleştirme: continuous integration
kesintisiz kurulum: continuous deployment
kesintisiz teslimat: continuous delivery
keşif sınaması: exploratory testing
keşifsel: exploratory
kimlik denetimi: authentication
kimlik numarası: tekil tanımlayıcı
kullanıcı topluluğu: user community
kullanmak: utilize
kurumsal veri yolu: enterprise service bus
kütük: registry
makale: paper
mikroservis: microservice
motosiklet sepeti: sidecar
nakil: transfer
nedensel: causal
nicel: quantitative

olay: event

ölçeklendirilmiş çeviklik: agility at scale

ölçüm: metric

örnekleyici: sampler

örüntü: pattern

öz ileti: self-message

özyinelemeli: recursive

parametre: parameter

paylaşımli kütüphane: shared library

sağlayıcı bağımlılığı: vendor lock-in

sağlayıcı bağımsız: vendor-neutral

sahip: owner

sarmalamak: encapsulate

sinama: test

sinama senaryosu: test case

sınırlandırılmış bağlam: bounded context

sınırlandırılmış kurum bağlamı: bounded organization context

sıralama diyagramı: sequence diagram

software visualization: yazılım görselleştirme

sunucusuz mimari: serverless computing

süreç: process

süreklilik: continuum

şablon: stereotype

takip: monitor

tasarım örüntüsü: design pattern

tezgah: bench

uçtan uca: end-to-end

uygulama programlama arayüzü: application programming interface

uygunluk: compliance

ürün sahipliği: product ownership

üst: parent

üstbilgi: header

üstveri: metadata

veri kalıcılığı katmanı: data persistency layer

web hizmeti: web service

yapılandırma: configuration

yapılanma: structuring

yaşam çizgisi: lifeline

yaşam süresi: lifespan

yayılım: propagation

yekpare: monolithic

yerel hafıza: local memory

yerleşim: layout

yokedici: destructor

yönlü: directed

yük: payload

yürütüm: execution

TEZDEN ÜRETİLMİŞ YAYINLAR

İletişim Bilgisi: bedrie@hotmail.com

Konferans Bildirileri

1. Eğrilmez, M.B. & Selçuk, Y.E., (2019). Utilization of Open Distributed Tracing Standards for Exploratory Testing and Visualization of Microservices. 6th International Symposium on Engineering, Artificial Intelligence & Applications, Girne, KKTC, Mart 2019

