T.C.
İSTANBUL KÜLTÜR ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

MİKROSERVİS EKOSİSTEMLERİNDE KAOS DENEYLERİNİN OTOMATİKLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Emrah ESEN

1900000474

Anabilim Dalı: Bilgisayar Mühendisliği
Programı : Bilgisayar Mühendisliği

Tez Danışmanı : Prof. Dr. Akhan Akbulut

HAZİRAN 2025

T.C.
İSTANBUL KÜLTÜR ÜNİVERSİTESİ
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

MİKROSERVİS EKOSİSTEMLERİNDE KAOS DENEYLERİNİN OTOMATİKLEŞTİRİLMESİ

YÜKSEK LİSANS TEZİ

Emrah ESEN

1900000474

Anabilim Dalı: Bilgisayar Mühendisliği
Programı : Bilgisayar Mühendisliği

Tez Danışmanı : Prof. Dr. Akhan Akbulut
Jüri Üyeleri : Prof. Dr. Özgür Koray Şahingöz
Doç. Dr. Fatma Patlar Akbulut

HAZİRAN 2025

ÖNSÖZ

Bu tez çalışmasının planlanması, yürütülmesi ve sonuçlandırılması sürecinde değerli bilgi ve deneyimleriyle rehberlik eden danışmanım Prof. Dr. Akhan Akbulut'a ve akademik katkılarıyla sürece destek veren eş danışmanım Prof. Dr. Çağatay Çatal'a saygı ve teşekkürlerimi sunarım.

HAZİRAN 2025

İÇİNDEKİLER

ÖNSÖZ ..... **i**
İÇİNDEKİLER ..... **ii**
KISALTMALAR ..... **iv**
TABLO LİSTESİ ..... **v**
ŞEKİL LİSTESİ ..... **vi**
ÖZET ..... **vii**
ABSTRACT ..... **ix**
1. GİRİŞ **1**
1.1 Problem Tanımı **1**
1.2 Tezin Amacı **2**
1.3 Tezin Organizasyonu **3**
2. TEMEL BİLGİLER VE LİTERATÜR TARAMASI **5**
2.1 Mikroservis Mimarisi **5**
2.1.1 Mikroservis Mimarisinin Avantajları **6**
2.1.2 Mikroservis Mimarisinin Dezavantajları **7**
2.2 Kaos Mühendisliği **8**
2.2.1 Temel İlkeler ve Prensipler **9**
2.2.2 Kaos Deneylerinde Kullanılan Araçlar **10**
2.3 Kaos Deneylerinin Mikroservis Mimarisinde Kullanımı **11**
2.4 Endüstriyel Örnekler **12**
2.5 Literatür taraması **13**
3. YÖNTEM **15**
3.1 Sistematik Literatür Taraması **16**
3.1.1 Kaos mühendisliği, yazılım sistemlerinin dayanıklılığını artırmak için üretim ortamlarında nasıl etkin bir şekilde uygulanır? **17**
3.1.2 Kaos deneyleri için hangi platformlar kullanıldı ? **19**
3.1.3 Kaos mühendisliği, sistem esnekliğini artırmada başarılı bir şekilde uygulanmasını sağlamak için mikro hizmet mimarilerine nasıl etkili bir şekilde uygulanır? **20**
3.1.4 Kaos mühendisliğinin merkezi olarak sağlanması, karmaşık sistemler genelinde Kaos deneylerinin yönetimini ne ölçüde etkili bir şekilde kolaylaştırabilir? **21**
3.1.5 İlgili makalelerde bildirilen zorluklar nelerdir? **22**
3.2 Deney Ortamı **23**
3.3 Kullanılan Araç ve Teknolojiler **24**
3.3.1 Minikube **24**
3.3.2 Chaos Mesh **25**

3.3.3 Jenkins **26**
3.3.4 Apache JMeter **26**
3.3.5 Diğer Bileşenler ve Altyapı: **26**
3.4 Kaos Deneylerinin Tasarlanması **26**
3.4.1 Pod Terminasyonu **27**
3.4.2 Ağ Gecikmesi **27**
3.4.3 Ağ Kesintisi **28**
3.4.4 Cpu ve Bellek Stres **28**
3.5 Veri Toplama ve Analiz Yöntemleri **29**
4. DENEYLERİN SONUÇLARI **31**
4.1 Pod Terminasyon Deney Sonuçları **31**
4.2 Ağ Gecikmesi Deney Sonuçları **35**
4.3 Ağ Kesintisi Deney Sonuçları **37**
4.4 Cpu Stres Deney Sonuçları **40**
4.5 Bellek Stres Deney Sonuçları **42**
4.6 Deneylerin Yürütülmesi **44**
5. **TARTIŞMA** **46**
5.1 Geçerliliğe Yönelik Şartlar Tehditler **46**
6. SONUÇ **48**
KAYNAKÇA **50**
EK 1. SLR ÇALIŞMASINDA SEÇİLEN YAYINLAR **54**

KISALTMALAR

SLR	: Sistematik Literatür Taraması (Systematic Literature Review)
CPU	: Merkezi İşlem Birimi
VPS	: Sanal Özel Sunucu
YAML	: Yet Another Markup Language
JSON	: JavaScript Object Notation
AWS EC2	: Amazon Web Servisleri Esnek Hesaplama Bulutu
GitOps	: Git-based Operations (Git Tabanlı Operasyonlar)
SLA	: Service Level Agreement (Hizmet Seviyesi Anlaşması)
FMEA	: Failure Mode and Effects Analysis (Hata Türleri ve Etkileri Analizi)
FTA	: Fault Tree Analysis (Arıza Ağacı Analizi)
CHAZOP	: Computational Hazard and Operability Study (Hesaplamalı Tehlike ve İşletilebilirlik İncelemesi)
CI/CD	: Cont. Int. / Cont. Deployment (Sürekli Entegrasyon / Sürekli Dağıtım)

TABLO LİSTESİ

Tablo 3.1. Araştırma soruları 15

ŞEKİL LİSTESİ

Şekil2.1 Mikroservis Mimarisi 5
Şekil 3.1 SLR İnceleme Protokolü 16
Şekil 3.2. Seçilen yayınların veritabanlarına göre dağılımı 17
Şekil 3.3 Açık Kaynak Kodlu Açık Ticaret Platformu Mimarisi 23
Şekil 3.4 ChaosMesh Mimari Yapısı 25
Şekil 4.1 Pod Terminasyon Sistem Cevap Süreleri 32
Şekil 4.2 Pod Terminasyon Servis Hata Sayıları 33
Şekil 4.3 Pod Terminasyon Servis Etki Analizi 34
Şekil 4.4 Ağ Gecikmesi Sistem Ortalama Cevap Süreleri 35
Şekil 4.5 Ağ Gecikmesi Servis Bazında Ortalama Cevap Süreleri 36
Şekil 4.6 Ağ Gecikmesi Servis Etki Analizi 37
Şekil 4.7 Ağ Kesintisi Sistem Ortalama Cevap Süreleri 38
Şekil 4.8 Ağ Kesintisi Servis Bazlı Hata Sayıları 39
Şekil 4.9 Ağ Kesintisi Servis Etki Analizi 40
Şekil 4.10 Cpu Stres Sistem Ortalama Cevap Süreleri 41
Şekil 4.11 Cpu Stres Servis Etki Analizi 42
Şekil 4.12 Bellek Stres Sistem Ortalama Cevap Süreleri 43
Şekil 4.13 Bellek Stres Servis Etki Analizi 44
Şekil 4.14 Pipeline Kurgusu 45

Üniversite : **T.C. İstanbul Kültür Üniversitesi**

Enstitü : **Lisansüstü Eğitim Enstitüsü**

Anabilim Dalı : **Bilgisayar Mühendisliği**

Program : **Bilgisayar Mühendisliği Tezli YL**

Tez Danışmanı : **Prof. Dr. Akhan Akbulut**

Tez Türü ve Tarihi : **Yüksek Lisans – Haziran 2025**

ÖZET

MİKROSERVİS EKOSİSTEMLERİNDE KAOS DENEYLERİNİN OTOMATİKLEŞTİRİLMESİ

Mikroservis mimarisi yazılım dünyasında devrim yaratmış, bireysel servislerin bağımsız olarak geliştirilip dağıtılmasını mümkün kılarak büyük ve monolitik uygulamaların dezavantajlarını ortadan kaldırmaktadır. Bu mimari yaklaşım sayesinde, sistemlerin ölçeklenebilirliği, esnekliği ve hızlı geliştirilmesi gibi önemli avantajlar elde edilmektedir. Ancak, mikroservislerin getirdiği bu avantajlar, aynı zamanda sistemlerin karmaşıklığını da artırarak sistemlerin beklenmedik hata ve arızalara karşı nasıl davrandığının anlaşılmasını zorlaştırmaktadır. Bu bağlamda, kaos mühendisliği, sistemlerin bu tür beklenmedik durumlara direncini artırmak ve potansiyel zayıf noktaları proaktif bir şekilde belirlemek için hayati öneme sahiptir.

Kaos deneylerinin uygulanması, sistemlerin gerçek dünya koşullarında nasıl performans gösterdiğinin derinlemesine anlaşılmasını sağlar. Ancak, bu deneylerin manuel olarak planlanması ve uygulanması, zaman ve kaynak yoğun bir süreçtir. Otomatikleştirme, bu süreci daha etkin ve verimli hale getirerek, kaos deneylerinin daha geniş bir çapta ve daha sık bir şekilde uygulanmasına olanak tanır. Bu çalışma, mikroservis ekosistemlerinde kaos deneylerinin otomatikleştirilmesine odaklanmaktadır, bu sayede sistemlerin beklenmedik durumlar karşısında nasıl daha dirençli hale getirilebileceğini araştırmaktadır.

Bu yaklaşımın, mikroservis tabanlı sistemlerin güvenilirliğini ve dayanıklılığını artırarak, yazılım geliştirme pratiklerinde önemli bir dönüşüm yaratması beklenmektedir. Bu tez kapsamında, bir e-ticaret uygulamasının mikroservis mimarisinde pod terminasyonu, ağ gecikmesi ve ağ kesintisi gibi kritik hata senaryolarına yönelik kaos deneyleri tasarlanıp uygulanmış, sistemin bu hata türlerine karşı dayanıklılığı ve hata yönetimi kabiliyeti kapsamlı şekilde analiz edilmiştir.

Anahtar Kelimeler: Mikroservis, Kaos Mühendisliği, Otomatik Hata Enjeksiyonu , Sistem Dayanıklılığı

University	**: T.C. İstanbul Kültür University**
Institute	**: Institute Of Graduate Studies**
Department	**: Computer Engineering**
Program	**: Computer Engineering**
Thesis Advisor	**: Prof. Dr. Akhan Akbulut**
Degree Awarded And Date	**: MA – June 2025**

ABSTRACT

AUTOMATING CHAOS EXPERIMENTS IN MICROSERVICES ECOSYSTEMS

Microservice architecture has transformed modern software development by enabling the independent development and deployment of discrete services, thereby addressing the limitations of large, monolithic systems. This architectural approach brings significant advantages such as enhanced scalability, flexibility, and faster development cycles. However, these benefits come with increased system complexity, which makes it more challenging to understand how services behave under unexpected conditions or failures. In this context, chaos engineering plays a crucial role in improving system resilience by proactively identifying potential vulnerabilities and testing the system's ability to withstand adverse scenarios.

Conducting chaos experiments allows for a deeper understanding of how systems behave under real-world conditions. Nevertheless, manually designing and executing these experiments often demands considerable time and resources. Automation offers a practical solution by streamlining the process, enabling more frequent and large-scale execution of chaos scenarios. This study focuses on automating chaos experiments within microservice-based ecosystems, aiming to explore how such systems can be made more resilient to unexpected disruptions. The proposed approach is expected to contribute to a shift in software engineering practices by enhancing the reliability and robustness of microservice architectures. In this thesis, chaos experiments focusing on critical failure scenarios namely pod

termination, network latency, and network latency were systematically designed and executed within the microservice architecture of an e-commerce application. A comprehensive analysis was conducted to evaluate the system's resilience and fault management capabilities in response to these failure modes.

Keywords: Microservice, Chaos Engineering, Automated Fault Injection, System Resilience

1. GİRİŞ

Mikroservis mimarileri, son yıllarda yazılım geliştirme alanında ön plana çıkan bir paradigma haline gelmiştir. Bağımsız hizmetlerin bir araya gelerek büyük ve karmaşık sistemleri oluşturduğu bu yaklaşım, geliştirme süreçlerinde esneklik ve ölçeklenebilirlik sağlar. Ancak, sistemlerin artan karmaşıklığı, potansiyel hata ve arızalara karşı dayanıklılığın sürekli olarak test edilmesini gerektirmektedir. Bu bağlamda, kaos mühendisliği, sistemlerin beklenmedik durumlara karşı nasıl tepki vereceğinin anlaşılmasına yardımcı olmak için kritik bir rol oynar. Kaos mühendisliği, sistemlerin dayanıklılığını ve hata toleransını artırmak amacıyla tasarlanmış, kontrollü deneyler yoluyla potansiyel zayıf noktaları proaktif bir şekilde tespit etmeyi amaçlar. Ancak, kaos deneylerinin manuel olarak yürütülmesi zahmetli ve kaynak yoğun bir süreçtir. Otomatikleştirme, bu süreci kolaylaştırarak, kaos deneylerinin daha geniş bir çapta ve daha sık bir şekilde uygulanmasını sağlar. Böylece, mikroservis ekosistemlerinin beklenmedik hata ve arızalara karşı daha dirençli hale getirilmesi amaçlanmaktadır.

Mikroservis ekosistemlerinde kaos deneylerinin otomatikleştirilmesine odaklanarak, bu yöntemin sistemlerin güvenilirliğini ve dayanıklılığını nasıl artırabileceğini incelemeyi hedeflemektedir. Otomatikleştirme süreçleri, araçları ve yöntemleri detaylı bir şekilde ele alınacak, mikroservis tabanlı sistemlerin dayanıklılığını artırma yönünde stratejik öneriler sunulacaktır. Bu çalışmanın, yazılım geliştirme pratiklerinde önemli bir dönüşüm yaratması ve mikroservis mimarilerinin potansiyelini daha da ortaya çıkarması beklenmektedir.

1.1 Problem Tanımı

Mikroservis mimarisi, günümüz yazılım geliştirme süreçlerinde önemli bir yer elde etmiştir. Monolitik mimarilere karşı sunduğu esneklik ve ölçeklenebilirlik gibi avantajlar, büyük ölçekli uygulamaların geliştirilmesinde bu yaklaşımı cazip kılmaktadır. Ancak, mikroservis tabanlı sistemlerde, birbirine bağımlı servislerin varlığı önemli karmaşıklıklar yaratmakta, bu da sistemin genel dayanıklılığını tehdit

eden zayıflıklar ortaya çıkarmaktadır. Servislerin arasındaki karmaşık etkileşimler, bir serviste meydana gelen arızanın diğer servislere yayılmasına neden olabilmekte, sonuçta kademeli arızalar ve kullanıcı deneyiminde ciddi bozulmalar yaşanmaktadır. Mikroservis tabanlı sistemlerde, birbirine bağımlı onlarca hatta yüzlerce küçük servis bulunmaktadır. Bu servisler arasındaki karmaşık etkileşimler ve ağ tabanlı iletişim, sistemin genel dayanıklılığını tehdit eden kritik noktalar oluşturmaktadır. Bir serviste meydana gelen arıza, çağrı zinciri boyunca diğer servislere yayılarak kademeli arızalara neden olabilmektedir. Bu durum, sistemin tamamının etkilenmesine ve kullanıcı deneyiminin ciddi şekilde bozulmasına yol açmaktadır.

Geleneksel test yaklaşımları, mikroservis mimarisinin bu karmaşık yapısını tam anlamıyla değerlendirmede yetersiz kalmaktadır. Birim testler, entegrasyon testleri ve uçtan uca testler gibi konvensiyonel test yöntemleri, genellikle kontrollü ortamlarda gerçekleştirilmekte ve canlı ortamındaki gerçek koşulları simüle edememektedir. Özellikle ağ gecikmeleri, kaynak tükenmesi, servis kesintileri ve beklenmedik yük artışları gibi canlı ortamda sıklıkla karşılaşılan durumlar, bu geleneksel test yaklaşımlarıyla öngörülememektedir. Bu bağlamda, mikroservis sistemlerinin dayanıklılığını proaktif olarak test edebilecek ve gerçek prodüksiyon koşullarını simüle edebilecek yenilikçi yaklaşımlara ihtiyaç duyulmaktadır. Mevcut test stratejilerinin bu karmaşık sistemlerin dayanıklılık gereksinimlerini karşılamada yetersiz kalması, sistemlerin canlı ortam üzerinde beklenmedik arızalarla karşılaşma riskini artırmakta ve bu durum hem işletmeler hem de son kullanıcılar açısından önemli mali ve operasyonel kayıplara neden olabilmektedir.

1.2 Tezin Amacı

Mikroservis mimarisi, günümüz yazılım geliştirme süreçlerinde önemli bir yer elde etmiştir. Monolitik mimarilere karşı sunduğu esneklik ve ölçeklenebilirlik gibi avantajlar, büyük ölçekli uygulamaların geliştirilmesinde bu yaklaşımı cazip kılmaktadır [1]. Ancak, mikroservis tabanlı sistemlerde, birbirine bağımlı servislerin varlığı önemli karmaşıklıklar yaratmakta, bu da sistemin genel dayanıklılığını tehdit eden zayıflıklar ortaya çıkarmaktadır. Servislerin arasındaki karmaşık etkileşimler, bir serviste meydana gelen arızanın diğer servislere yayılmasına neden olabilmekte, sonuçta kademeli arızalar ve kullanıcı deneyiminde ciddi bozulmalar yaşanmaktadır [2].

Bu çalışma, Kaos Mühendisliği metodolojisinin mikroservis mimarisinde nasıl etkili bir şekilde uygulanabileceğini araştırmak ve bu amaçla sistematik bir yaklaşım geliştirmek amacıyla tasarlanmıştır. ChaosMesh platformu kullanılarak çeşitli kaos deneyleri tasarlanacak ve Jenkins CI/CD pipeline'ı aracılığıyla bu deneylerin otomatik olarak yürütülmesi sağlanacaktır. JMeter aracılığıyla oluşturulan sentetik trafik altında, sistemin performans metrikleri, yanıt süreleri ve hata oranları gibi kritik göstergelerin izlenmesi ve analiz edilmesi planlanmaktadır. Çalışmanın ana hedefleri arasında ağ kesintileri, pod terminasyonları, kaynak kısıtlamaları ve servis gecikmelerine karşı sistemin davranışının test edilmesi ve sistemdeki zayıf noktaların tespit edilmesi bulunmaktadır. Aynı zamanda, dayanıklılık testlerinin sürekli entegrasyon süreçlerine dahil edilmesi yoluyla organizasyonlarda sürekli iyileştirme kültürünün geliştirilmesine katkı sağlanması amaçlanmaktadır.

Bu tez çalışması, Kaos Mühendisliği alanında yapılan sistematik literatür taraması bulgularının gerçek bir uygulama üzerinde test edilmesi ve doğrulanması yoluyla hem akademik hem de pratik katkılar sağlamayı hedeflemektedir. Çalışma sonucunda, mikroservis mimarilerine sahip sistemlerin dayanıklılığının sistematik olarak değerlendirilmesi ve artırılması için bilimsel temellere dayanan, endüstride uygulanabilir bir yaklaşım geliştirilmesi amaçlanmaktadır.

1.3 Tezin Organizasyonu

Bu çalışma, mikroservis mimarilerinde kaos deneylerinin otomatikleştirilmesini ele almakta ve konuya bütüncül bir bakış açısı sunmayı hedeflemektedir [3]. Tez, toplamda altı ana bölümden oluşmaktadır. İlk bölümde, çalışmanın amacı, problemi ve temel yaklaşımı özetlenmiştir. Burada, araştırmanın neden gerekli olduğu, hangi boşluğu doldurmayı hedeflediği ve genel hatlarıyla ele alınan konular okuyucuya sunulmaktadır [4]. İkinci bölümde, mikroservis mimarisi ve kaos mühendisliğiyle ilgili temel kavramlar detaylı bir şekilde ele alınmıştır. Mikroservislerin yapısı, avantajları ve karşılaşılan zorluklar aktarıldıktan sonra, kaos mühendisliğinin temel prensipleri, tarihsel gelişimi ve bu alanda kullanılan araçlar üzerinde durulmuştur. Ayrıca, literatürdeki güncel çalışmalar ve uygulama örnekleri de bu bölümde incelenmiştir.

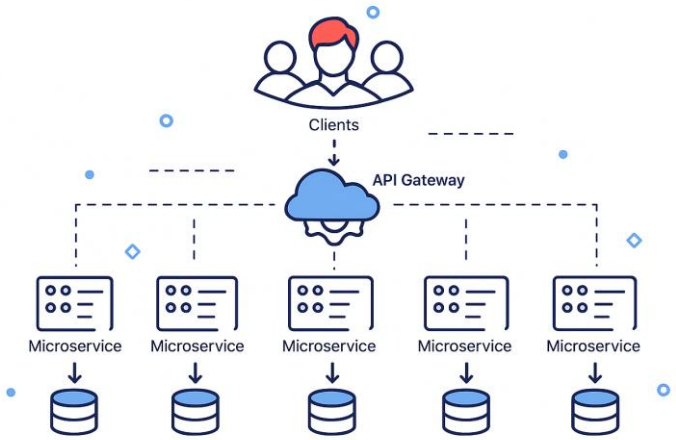
Üçüncü bölümde, tezde yürütülen kaos deneylerinin tasarımı, deney ortamı ve kullanılan araçlar anlatılmıştır. Deneylerin gerçekleştirildiği altyapının özellikleri, kullanılan otomasyon araçları ve izlenen metodoloji adım adım açıklanmıştır. Deneylerin sistematik şekilde planlanıp uygulanması için izlenen süreç detaylandırılmıştır. Dördüncü bölümde, gerçekleştirilen deneyler ve elde edilen bulgulara yer verilmiştir. Kaos deneylerinin sistem üzerindeki etkileri, farklı hata senaryoları altında ortaya çıkan sonuçlar ve bu sonuçların analizi sunulmuştur. Bulgular, ilgili grafik ve ölçümlerle desteklenmiş, elde edilen veriler ışığında değerlendirmeler yapılmıştır. Beşinci bölümde, elde edilen bulguların literatür ile karşılaştırılması ve yorumlanması gerçekleştirilmiştir. Bulguların geçerliliği, sınırlılıkları ve çalışmanın güvenilirliği tartışılmıştır. Ayrıca, uygulanan yöntemlerin güçlü ve zayıf yönleri değerlendirilmiş, olası tehditler ve geçerlilik kriterleri ele alınmıştır. Son bölümde ise genel bir değerlendirme ve sonuçlara yer verilmiştir. Çalışmanın katkıları özetlenmiş, mevcut yaklaşımların sektöre ve akademiye olası etkileri üzerinde durulmuştur. Ayrıca, ileride yapılabilecek çalışmalar için öneriler sunulmuştur.

2. TEMEL BİLGİLER VE LİTERATÜR TARAMASI

Bu bölümde tez kapsamındaki çalışmalarda kullanılan temel kavramlar ele alınmıştır. Bunlardan ilki modern dağıtık sistemlerin temelini oluşturan mikroservis mimarisidir. Mikroservis mimarisinin temel prensipleri, avantajları ve zorlukları detayları ile anlatılmıştır. İkinci olarak ise çalışmamıza temel oluşturan Kaos Mühendisliği disiplininin teorik temelleri, tarihsel gelişimi ve uygulama prensipleri ortaya konmuştur. Daha sonra sistem dayanıklılığı kavramına yönelik temel tanımlar olan dayanıklılık mühendisliği, hata toleransı ve dayanıklılık desenlerinin detayları anlatılmıştır. Son olarak kaos testi araçları, otomatize test yaklaşımları ve mevcut literatürdeki boşluklar kapsamlı bir şekilde incelenmiştir.

2.1 Mikroservis Mimarisi

Mikroservis mimarisi, geleneksel monolitik uygulama geliştirme yaklaşımına alternatif olarak ortaya çıkmış ve yazılım sistemlerinin küçük, bağımsız ve gevşek bağlı servisler halinde organize edilmesini öngören bir mimari yaklaşımdır. Bu mimari paradigma, büyük ve karmaşık uygulamaların yönetilebilir parçalara bölünmesini sağlayarak, geliştirme süreçlerinde esneklik ve ölçeklenebilirlik avantajları sunmaktadır [5][6].



Şekil2.1 Mikroservis Mimarisi

Mikroservis mimarisi, Martin Fowler ve James Lewis tarafından tanımlanan temel prensipler üzerine inşa edilmektedir. Bu prensipler arasında tek sorumluluk

ilkesi, servisler arası gevşek bağlılık ve her servisin kendi veri tabanına sahip olması gibi önemli kavramlar yer almaktadır [7]. Her mikroservis, kendi yaşam döngüsünü bağımsız olarak yönetebilme kabiliyetine sahiptir ve bu durum geliştirme, test etme, dağıtım ve ölçeklendirme süreçlerinin servis bazında gerçekleştirilebilmesini sağlamaktadır [8].

2.1.1 Mikroservis Mimarisinin Avantajları

Mikroservis mimarisi, büyük ve karmaşık yazılım sistemlerini küçük, bağımsız çalışan bileşenlere ayırarak yönetilebilirliği ve esnekliği artıran modern bir yazılım geliştirme yaklaşımıdır. Bu yaklaşımın birçok teknik ve organizasyonel avantajı bulunmaktadır [9]. Bağımsız ölçeklenebilirlik mikroservis mimarisinin temel özelliklerinden biridir. Her bir servis, kendi ihtiyaç duyduğu kaynaklara göre bağımsız olarak yatay veya dikey biçimde ölçeklenebilir. Böylece yalnızca yüksek yük altındaki bileşenlerin büyütülmesiyle sistem kaynakları daha etkin kullanılabilir ve maliyet açısından optimizasyon sağlanır [9]. Mikroservis yapısı ayrıca farklı yazılım dillerini ve veri tabanlarını aynı sistem içinde bir arada kullanma olanağı sağlayarak teknolojik çeşitliliği destekler. Bu durum, her servisin kendi gereksinimlerine uygun teknolojilerle geliştirilmesine imkan tanır ve sistem genelinde esneklik sunar [9].

Bu yapı, hızlı geliştirme ve dağıtım süreçlerine olanak tanır. Küçük ve bağımsız servisler, daha kısa geliştirme döngülerine sahiptir. Bu sayede kod değişiklikleri daha hızlı test edilip üretim ortamına alınabilir. Sürekli dağıtım süreçleri de bu yapıya doğal olarak entegre edilebilir [10]. Ekip özerkliği, mikroservislerin bir diğer önemli avantajıdır. Her bir servis için bağımsız geliştirme ekipleri atanabilir ve bu ekipler, servislerinin yaşam döngüsünden bütünüyle sorumlu olarak çalışabilir. Bu organizasyon yapısı, geliştirici ekiplerin çevikliğini artırır ve daha hızlı karar alma süreçlerini destekler [10]. Sistemlerin hata toleransını artıran bir diğer özellik ise hata izolasyonudur. Mikroservisler kendi başlarına çalıştıkları için, bir serviste yaşanan arıza diğer servisleri doğrudan etkilemez. Bu da sistem genelinin çökmesini engelleyerek kesintisiz hizmet sunumuna katkı sağlar [11]. Mikroservislerin küçük ve odaklanmış yapısı, sürekli entegrasyon ve sürekli dağıtım (CI/CD) süreçlerinin uygulanmasını da kolaylaştırır. Her servis bağımsız olarak test edilebilir, derlenebilir ve dağıtılabilir. Bu yapı, yazılım geliştirme sürecinde otomasyonu ve kalite kontrolünü teşvik eder. Yeniden kullanılabilirlik, bu mimarinin başka bir avantajıdır. Bir kez

geliştirilen servisler, benzer işlevselliğe ihtiyaç duyan başka sistemlerde veya projelerde kolayca tekrar kullanılabilir. Bu durum geliştirme süresini kısaltırken yazılım mimarisinde tutarlılığı da artırır [11]. Son olarak, bakım ve güncellemeler servis bazında gerçekleştirilebilir. Yalnızca ilgili servisin güncellenmesi yeterli olduğu için tüm sistemin durmasına gerek kalmaz. Bu da bakım sürelerini azaltır ve hizmet sürekliliğini garanti altına alır [11].

2.1.2 Mikroservis Mimarisinin Dezavantajları

Mikroservis mimarisi, birçok avantajının yanında, sistem karmaşıklığını ve yönetim maliyetini artıran çeşitli zorluklar da doğurmaktadır. Bu başlık altında, bu mimarinin uygulamada karşılaşılan başlıca dezavantajları değerlendirilecektir. İlk olarak, ağ gecikmeleri ve iletişim problemleri mikroservis tabanlı sistemlerin en yaygın sorunlarından biridir. Servislerin birbirleriyle yalnızca ağ üzerinden iletişim kurması, ağ üzerinde yaşanacak her türlü gecikmenin veya kopmanın sistem davranışını doğrudan etkileyebilmesine yol açar. Bu durum, sistemin performansını düşürebilir ve kullanıcı deneyimini olumsuz olarak etkileyebilir [12].

Yönetimsel karmaşıklık, mikroservislerin sayısı arttıkça daha belirgin hale gelir. Her servisin bağımsız olarak sürdürülmesi, versiyonlanması, yapılandırılması ve dağıtılması, sistemin bütünsel kontrolünü güçleştirebilir. Bu da zamanla hem operasyonel yükü artırır hem de koordinasyonu zorlaştırır [11]. Bir diğer önemli sorun veri tutarlılığıdır. Dağıtık veri yönetimi, özellikle tutarlılığın kritik olduğu uygulamalarda önemli bir zorluktur. Farklı servislerin farklı veri kaynakları kullanması ve bu verilerin eş zamanlı güncellenmesi gerektiğinde, tutarsızlık riski ciddi boyutlara ulaşabilir [12].

Mikroservis mimarilerinde hata ayıklama da oldukça zorlayıcıdır. Bir hatanın hangi servisten kaynaklandığını bulmak için çok sayıda log kaynağının ve izleme aracının entegre çalışması gerekir. Servislerin birbirine bağımlı olması, hataların zincirleme etkiyle yayılmasına ve kök nedenin saptanmasının gecikmesine neden olabilir [13]. Benzer şekilde, test süreçleri monolitik yapılara göre daha karmaşıktır. Tek bir bütün yerine, bağımsız çalışan birçok bileşenin entegrasyonunun test edilmesi gerekir. Bu da test otomasyonunu zorlaştırır ve daha karmaşık test senaryoları oluşturmayı gerektirir [11]. Güvenlik açısından, mikroservislerin her biri potansiyel

bir saldırı yüzeyi oluşturur. Kimlik doğrulama, yetkilendirme ve veri güvenliği gibi önlemlerin her servis için ayrı ayrı uygulanması gerekir. Bu da güvenlik yönetimini karmaşıklaştırır ve saldırı risklerini artırabilir [12]. Operasyonel yük, mikroservislerin bağımsız doğası nedeniyle daha fazladır. Servislerin sürekli olarak izlenmesi, güncellenmesi ve yönetilmesi; merkezi loglama, izleme ve yapılandırma araçlarının etkin kullanımını zorunlu kılar [11]. Son olarak, bu mimarinin başarılı bir şekilde uygulanabilmesi için uzman ekipler gereklidir. Mikroservislerin mimarisi, dağıtık sistemler, otomasyon araçları, konteyner orkestrasyon sistemleri gibi konularda derin bilgi ve deneyim sahibi ekiplerle yürütülmelidir [13].

2.2 Kaos Mühendisliği

Kaos Mühendisliği (Chaos Engineering), dağıtık sistemlerin üretim ortamında beklenmedik durumlarla karşılaştığında dayanıklılığını test etmek amacıyla kontrollü deneyler yapma disiplini olarak tanımlanmaktadır [3], [14]. Bu yaklaşım, sistemlerin gerçek dünyada karşılaşabileceği arıza senaryolarını önceden simüle ederek, potansiyel zayıflıkları ortaya çıkarmayı ve sistem güvenilirliğini artırmayı hedeflemektedir. Netflix tarafından 2008 yılında başlatılan bu disiplin, şirketin bulut altyapısına geçiş sürecinde sistemlerin dayanıklılığını test etme ihtiyacından doğmuştur. İlk kaos aracı olan "Chaos Monkey", AWS EC2 instance'larını rastgele kapatarak sistemin bu arızalara nasıl tepki verdiğini gözlemlemiştir. Bu yaklaşım, zamanla "Simian Army" adı verilen kapsamlı bir araç setine dönüşmüş ve Kaos Mühendisliği disiplininin temellerini oluşturmuştur.

Kaos Mühendisliği'nin temel felsefesi, sistemlerin arızalara karşı proaktif olarak hazırlanmasıdır. Geleneksel yaklaşımlar arızaları önlemeye odaklanırken, Kaos Mühendisliği arızaların kaçınılmaz olduğunu kabul eder ve sistemlerin bu durumlarla nasıl başa çıkacağını öğrenmeyi amaçlar. Bu yaklaşım, "antifragility" kavramıyla da ilişkilidir; yani sistemlerin stres altında daha güçlü hale gelmesi prensibini benimser.

Kaos Mühendisliği deneylerinin temel amacı, sistem davranışını anlayarak güven düzeyini artırmaktır. Bu deneylerin sistematik olarak yapılması, ekiplerin sistemlerinin sınırlarını öğrenmesini ve potansiyel sorunlara karşı hazırlıklı olmalarını sağlar. Aynı zamanda, bu yaklaşım organizasyonlarda dayanıklılık kültürünün gelişmesine de katkıda bulunur.

Modern mikroservis mimarilerinde Kaos Mühendisliği'nin önemi daha da artmıştır. Mikroservislerin karmaşık bağımlılık ağları, kademeli arızalar riskini artırmakta ve bu nedenle sistematik kaos testi yaklaşımları kritik hale gelmektedir. Konteyner orkestrasyon platformları olan Kubernetes gibi ortamlarda, Kaos Mühendisliği araçları sistemlerin gerçek koşullardaki davranışlarını test etmek için vazgeçilmez hale gelmiştir.

2.2.1 Temel İlkeler ve Prensipler

Kaos Mühendisliği'nin etkili bir şekilde uygulanabilmesi için belirli temel ilkeler geliştirilmiştir. Bu prensipler, Netflix tarafından formüle edilmiş ve Kaos Mühendisliği topluluğu tarafından kabul görmüştür [3], [12].

- Kararlı Durum Hipotezi: Sistemin normal çalışma koşullarında nasıl davranması gerektiğine dair ölçülebilir bir hipotez oluşturulur. Bu hipotez, yanıt süreleri, başarı oranları ve hata oranları gibi metriklere dayanır.
- Gerçek Dünya Olaylarının Simülasyonu: Chaos deneyleri, prodüksiyonda gerçekleşebilecek gerçek arıza senaryolarını simüle eder. Server çökmeleri, network kesintileri ve disk arızaları gibi durumlar test edilir.
- Prodüksiyon Ortamında Deney: En değerli sonuçlar, gerçek kullanıcı trafiği altında prodüksiyon ortamında yapılan deneylerden elde edilir. Test ortamları gerçek koşulları tam yansıtamaz.
- Sürekli Otomasyon: Chaos deneyleri otomatik olarak yürütülmeli ve CI/CD süreçlerine entegre edilmelidir. Bu, düzenli tekrarları ve sürekli izlemeyi sağlar.
- Etki Alanını Sınırlama: Deneylerin sistem üzerindeki etkisi kontrol edilmeli ve minimize edilmelidir. Küçük çaplı deneylerle başlayıp kademeli olarak genişletme yaklaşımı benimsenmelidir.
- Metrik Odaklı Yaklaşım: Chaos deneyleri objektif ve ölçülebilir sonuçlar üretmelidir. Somut metriklerle sistem davranışı analiz edilir.

Bu prensipler, Kaos Mühendisliği'nin güvenli ve sistematik bir şekilde uygulanmasını sağlar ve organizasyonların sistem dayanıklılığını sürekli iyileştirmelerine olanak tanır.

2.2.2 Kaos Deneylerinde Kullanılan Araçlar

Kaos Mühendisliği deneyleri için geliştirilmiş çeşitli araçlar ve platformlar bulunmaktadır. Yaptığımız sistematik literatür taramasında 38 farklı Kaos Mühendisliği aracı tespit edilmiştir. Bu araçlar arasında en sık kullanılanlar şunlardır:

- Chaos Monkey: Netflix tarafından geliştirilen ilk kaos aracıdır. AWS EC2 instance'larını rastgele kapatarak sistem dayanıklılığını test eder. Simian Army'nin bir parçası olarak alt yapı seviyesinde arızalar oluşturur [3].
- Gremlin: Hizmet olarak Kaos Mühendisliği sunan ticari platform. Alt yapı seviyesi ve uygulama seviyesi saldırılar gerçekleştirebilir. CPU tüketimi, disk doldurma, ağ gecikmeleri gibi çeşitli kaos türlerini destekler [11].
- ChaosMesh: Kubernetes ortamında kaos deneyleri yapmak için geliştirilmiş açık kaynak platform. Pod sonlandırma, ağ gecikmeleri, I/O hataları, DNS hataları gibi çeşitli kaos türlerini destekler [15].
- Litmus: Cloud-native Kaos Mühendisliği için geliştirilmiş kubernetes tabanlı açık kaynak platform. İş akışı tabanlı kaos deneyleri ve GitOps entegrasyonu sunar [16].
- Chaos Toolkit: Python tabanlı açık kaynak Kaos Mühendisliği platformu. Çeşitli sistemler için genişletilebilir sürücüler içerir ve JSON/YAML formatında deney tanımlamasına olanak sağlar [17].
- Pumba: Docker container'ları için geliştirilmiş kaos test aracı. Container'ları durdurma, ağ trafiğini manipüle etme ve kaynak sınırlaması gibi özellikler sunar [15].
- ToxiProxy: Ağ seviyesinde kaos oluşturmak için geliştirilmiş proxy aracı. Gecikme, bant genişliği sınırlaması, bağlantı kesintileri gibi ağ problemlerini simüle eder [17].

Bu araçlar, alt yapı, platform, uygulama gibi farklı seviyeler ve pod termination, network chaos, resource stress gibi farklı kaos türleri için özelleşmiş çözümler sunmaktadır.

2.3 Kaos Deneylerinin Mikroservis Mimarisinde Kullanımı

Mikroservis mimarisinde Kaos Mühendisliği'nin uygulanması, monolitik sistemlere kıyasla farklı yaklaşımlar ve özel dikkat gerektiren konular içermektedir. Mikroservislerin dağıtık doğası ve karmaşık bağımlılık ağları, kaos deneyleri için hem yeni fırsatlar hem de ek zorluklar yaratmaktadır [13]. Mikroservis sistemlerinde Kaos Mühendisliği'nin temel amacı, servisler arası etkileşimlerin dayanıklılığını test etmek ve kademeli arızaların önlenmesini sağlamaktır. Yaptığımız sistematik literatür taramasında belirtildiği üzere, her mikroservis bağımsız olarak deploy edilebilir ve ölçeklendirilebilir olduğu için, kaos deneyleri de servis bazında granüler olarak tasarlanabilir. Mikroservis mimarisinde hata enjekte etme testi yaklaşımı özellikle önemlidir. Bu yöntem, sistem içindeki çeşitli arıza modlarını simüle ederek ağ sorunları, servis kesintileri ve kaynak sıkıntıları gibi durumları test eder [18]. Hipotez odaklı deneyler yaklaşımı ile sistem davranışı hakkında önceden belirlenen hipotezler test edilir ve sistem dayanıklılığı değerlendirilir [19].

Mikroservis tabanlı kaos mühendisliğinde blast radius yönetimi (etki alanı/bölgesi yönetimi) kritik öneme sahiptir. Başlangıçta küçük deneylerle başlayarak etkinin anlaşılması ve ardından kapsamın genişletilmesi yaklaşımı benimsenmelidir. Bu sayede beklenmedik sistem geneli etkilerden kaçınılır ve kontrollü deneyler gerçekleştirilebilir. Dayanıklılık gereksinimlerinin ortaya çıkarılması süreci, mikroservis mimarisinin özgül ihtiyaçlarını belirlemek için kullanılır. Bu süreç, her mikroservisin ve servisler arası etkileşimlerin dayanıklılık gereksinimlerini tanımlamayı içerir. Sürekli test ve iyileştirme yaklaşımı ile kaos deneyleri düzenli olarak tekrarlanır ve sistem zayıflıkları proaktif olarak tespit edilir.

Gözlemlenebilirlik ve iyileştirme entegrasyonu, mikroservis kaos deneyleri sırasında sistem davranışının gerçek zamanlı izlenmesini sağlar. Bu yaklaşım, arızalara karşı etkili iyileştirme stratejilerinin geliştirilmesine ve genel sistem dayanıklılığının artırılmasına katkıda bulunur. Mikroservis mimarisinde Kaos Mühendisliği uygulamaları, sistemdeki güvenlik açıklarını ve risk faktörlerini ortaya çıkararak hata toleransı ve sağlamlık seviyelerinin artırılmasını sağlar [14]. Bu deneylerin sistematik olarak uygulanması, mikroservis ekosistemlerinin canlı ortamındaki beklenmedik durumlarla başa çıkma kabiliyetlerini önemli ölçüde geliştirir [20].

2.4 Endüstriyel Örnekler

Kaos mühendisliği, özellikle büyük ölçekli ve yüksek erişilebilirlik gerektiren sistemlerde yaygın olarak uygulanmaktadır. Bu yaklaşımın en bilinen ve öncü örneği Netflix tarafından gerçekleştirilen Chaos Monkey ve Simian Army uygulamalarıdır. Netflix, mikroservis tabanlı altyapısında yaşanabilecek çeşitli arızaları (örneğin; sunucu çökmesi, ağ gecikmesi, disk doluluğu vb.) gerçek üretim ortamında simüle ederek, sistemin bu tür beklenmeyen koşullarda nasıl davrandığını sürekli olarak test etmektedir. Bu sayede, potansiyel zayıflıklar proaktif olarak tespit edilmekte ve üretim ortamında büyük çaplı kesintilerin önüne geçilmektedir. Netflix'in yaptığı bu çalışmalar sayesinde, hata toleransı ve hizmet sürekliliğinde ciddi artışlar sağlanmıştır.

Amazon, Google ve Microsoft gibi diğer büyük teknoloji şirketleri de benzer şekilde kaos mühendisliği prensiplerini uygulamaktadır. Amazon, "GameDay" isimli tatbikatlarla, ekiplerin kriz anlarında sistemin yeniden ayağa kaldırılması süreçlerini ve hata senaryolarına karşı hazırlıklarını test etmektedir. Google, "DiRT" (Disaster Recovery Testing) programı ile veri merkezi ölçeğinde kesinti simülasyonları yaparak sistemlerinin ve çalışanlarının felaket anındaki dayanıklılığını artırmayı hedeflemektedir. Bunun yanında, finans, perakende ve bulut servis sağlayıcıları gibi farklı sektörlerde de Kaos Mühendisliği uygulamalarının başarılı örnekleri mevcuttur. Örneğin; Groupons, kendi geliştirdiği otomasyon aracı ile mikroservis mimarisinde çeşitli hata senaryolarını otomatik olarak enjekte ederek, sistemin genel hata toleransını ve toparlanma hızını artırmıştır. Benzer şekilde, Gremlin aracı, mikroservis mimarilerinde sistematik dayanıklılık testleri için birçok büyük ölçekli şirket tarafından kullanılmaktadır ve özellikle müşteri deneyimini etkileyebilecek potansiyel darboğazlar proaktif olarak tespit edilmektedir.

Endüstride Kaos Mühendisliği uygulamalarının sağladığı başlıca başarılar arasında şunlar öne çıkmaktadır:

- Arıza Süresinde Azalma: Kaos deneyleri sayesinde kritik hatalar önceden tespit edilip düzeltilmiş, üretim ortamında yaşanan arıza süreleri belirgin şekilde azaltılmıştır.
- Hizmet Sürekliliğinde Artış: Sistemler, beklenmedik arızalara karşı daha dayanıklı hale gelmiş, SLA değerlerinde iyileşmeler görülmüştür.

- Operasyonel Maliyetlerde Azalma: Proaktif hata tespiti sayesinde ani kesintilerin yol açtığı iş kaybı ve müdahale maliyetleri minimize edilmiştir.
- Kültürel Dönüşüm: Ekipler hata korkusunu yenmiş ve "her şeyin bir gün bozulabileceği" anlayışıyla yazılım geliştirme ve operasyon süreçlerini sürekli olarak geliştirmiştir.

Sonuç olarak, kaos mühendisliği pratiklerinin endüstrideki uygulamaları, yalnızca sistemlerin teknik dayanıklılığını artırmakla kalmamış, aynı zamanda organizasyonların operasyonel olgunluğunu ve müşteri memnuniyetini de önemli ölçüde artırmıştır.

2.5 Literatür taraması

Son yıllarda mikroservis mimarisi ve Kaos Mühendisliği konuları yazılım mühendisliği literatüründe önemli bir yer edinmiştir. Dağıtık sistemlerin artan karmaşıklığıyla birlikte, bu sistemlerin dayanıklılığını ve hata toleransını ölçmek ve iyileştirmek için yeni test yöntemlerine olan ihtiyaç daha belirgin hale gelmiştir. Kaos Mühendisliği, sistemlerin üretim ortamında rastgele veya planlı hatalar enjekte edilerek beklenmedik durumlara karşı dayanıklılığının ölçülmesini amaçlar. Netflix'in Simian Army ve Chaos Monkey araçları bu alandaki öncülerden olup, birçok farklı uygulama ve mikroservis mimarisi üzerinde test edilmiştir [14], [21]. Kaos Mühendisliği uygulamalarının mikroservis tabanlı mimarilerde etkin şekilde kullanılabilmesi için deneylerin otomatikleştirilmesi gerekmektedir. Literatürde, hata enjekte etme, risk tabanlı deney seçimi ve deneylerin otomatik yürütülmesi üzerine çeşitli yaklaşımlar önerilmektedir [16], [22].

Mikroservis tabanlı sistemlerde karşılaşılan başlıca zorluklar, bağımsız servislerin çokluğu, servisler arası karmaşık iletişim ve arızaların kök nedeninin hızlıca bulunamamasıdır [13], [23]. Kaos mühendisliği alanında geliştirilen Chaos Mesh, Chaos Monkey, Gremlin, Litmus gibi araçlar sayesinde; üretim ortamında ya da kontrollü test ortamlarında servislerin dayanıklılığı sistematik olarak test edilebilmektedir. Bazı çalışmalar, otomatik kaos deneylerinin risk analiz teknikleriyle (ör. FMEA, FTA, CHAZOP) entegre edilmesini önermektedir. Bu sayede deneylerin

sistem üzerindeki etkileri önceden tahmin edilerek, en kritik noktaların test edilmesi sağlanmaktadır [10], [22].

Ayrıca deneylerin sadece üretim ortamında değil, simülasyon ortamlarında da gerçekleştirilmesi için çeşitli platformlar geliştirilmiştir (Chaos Mesh, Gremlin, Litmus, Chaos Monkey vb.) [11], [21]. Sonuç olarak, literatürde hem endüstriyel hem akademik açıdan Kaos Mühendisliği uygulamalarının mikroservis mimarilerinde sistematik olarak uygulanabilmesi için; deneylerin otomatikleştirilmesi, sonuçların analiz edilmesi ve elde edilen çıktılar doğrultusunda sistemin sürekli iyileştirilmesi yönünde önemli ilerlemeler kaydedilmiştir. Bu alandaki güncel çalışmalar, otomasyon seviyesinin artırılması, daha gerçekçi hata modellemeleri ve deneylerin minimum riskle uygulanması üzerine yoğunlaşmaktadır.

3. YÖNTEM

Bu bölümde, mikroservis mimarisinde kaos mühendisliğinin otomatikleştirilmesi kapsamında belirlenen temel araştırma soruları ile bu sorulara karşı literatürde bulunan veya önerilen çözüm yaklaşımları açıklanmıştır. Araştırma kapsamında ele alınan ve cevaplanması gereken beş temel soru Tablo 3.1'de sunulmaktadır. Bu sorulardan ilk dördünün yanıtlarını bulmak amacıyla sistematik literatür taraması (Systematic Literature Review - SLR) gerçekleştirilmiş ve ilgili yayınlar detaylı şekilde incelenmiştir. Literatürde yapılan çalışmalar bir yandan bu tez çalışmasına yön verirken, diğer yandan çalışma sonunda elde edilen bulguların karşılaştırılmasına olanak sağlamıştır. Bazı soruların yanıtlanmasında doğrudan literatürde yer alan kaynaklardan faydalanmak, tez çalışmasının özgün katkılarına daha fazla odaklanılmasını mümkün kılmıştır.

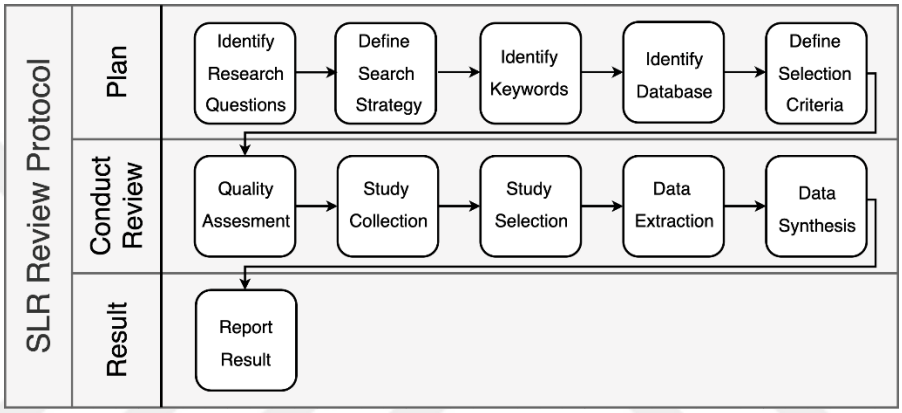
Tablo 3.1. Araştırma soruları

	Araştırma Soruları
S1	Kaos Mühendisliği, yazılım sistemlerinin dayanıklılığını artırmak için üretim ortamlarında ne şekilde etkili olarak uygulanmaktadır?
S2	Kaos deneyleri için hangi platformlar kullanılmıştır?
S3	Kaos Mühendisliği, mikroservis mimarilerine sistem dayanıklılığını artıracak şekilde başarılı bir şekilde nasıl entegre edilmektedir?
S4	Kaos Mühendisliğinin merkezi olarak yönetilmesi, karmaşık sistemlerde deneylerin yönetimini ne ölçüde kolaylaştırmaktadır?
S5	İlgili makalelerde bildirilen zorluklar nelerdir?

Yapılan SLR çalışması ile araştırma sorularından ilk dördüne sistematik olarak yanıt verilmiştir. SLR çalışması sonucunda elde edilen bulgular aşağıda ayrıntılı olarak sunulmaktadır. Ayrıca, çalışmada karşılaşılan zorluklar ve çözüm önerileri de tartışılmıştır.

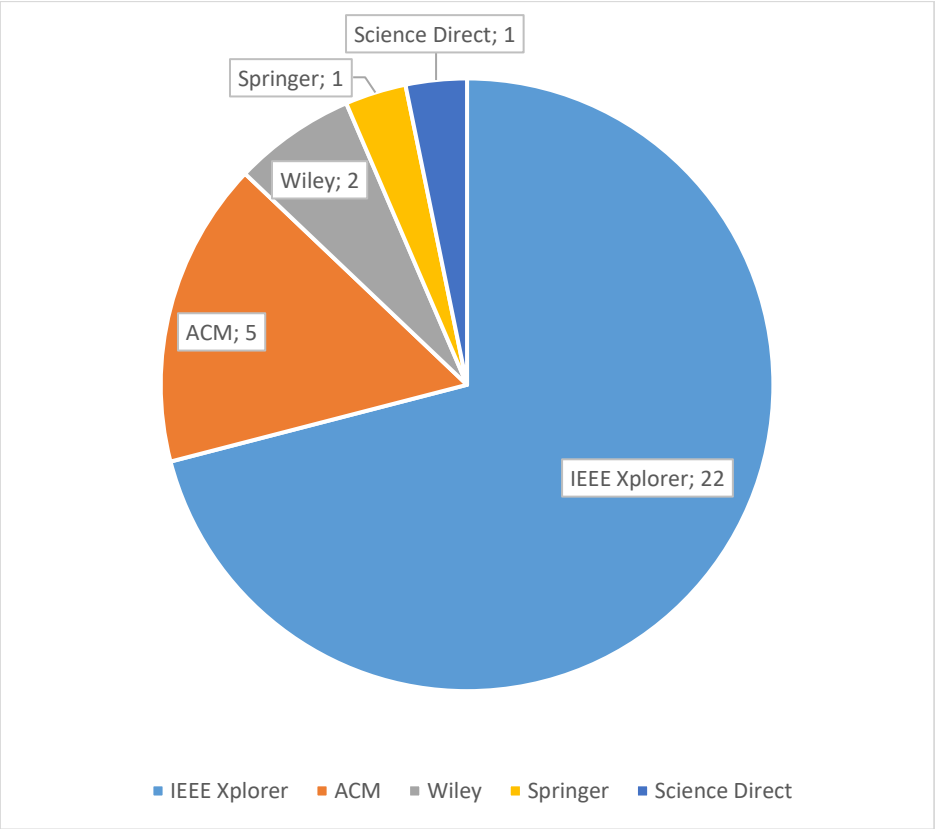
3.1 Sistematik Literatür Taraması

Sistematik literatür taraması kapsamında, kaos mühendisliği ve mikroservis mimarileri üzerine yapılan çalışmalarda karşılaşılan temel zorluklar, çözüm yöntemleri ve deneyimler analiz edilmiştir. Bu çalışmada Şekil3.1 de yer alan SLR inceleme protokol takip edilmiştir. İlk aşamada, Science Direct, IEEE Xplore, Scopus ve Springer gibi yaygın kullanılan akademik veri tabanlarında "chaos engineering", "chaos experiments", "microservices" gibi anahtar kelimelerle kapsamlı bir tarama yapılmıştır. Elde edilen çalışmalar, başlık, özet ve anahtar kelimelerine göre ön elemeden geçirilmiştir.



Şekil 3.1 SLR İnceleme Protokolü

İkinci aşamada, belirlenen ön inceleme sonrası, çalışmalara tam metin erişim sağlanmış, dil olarak İngilizce olması ve deneysel sonuç içermesi ana seçim kriterleri olarak uygulanmıştır. Son 10 yıl içerisinde yayımlanmış makaleler dikkate alınmış ve belirli kriterlere göre toplam 31 yayın seçilmiştir. Şekil 3.2'de seçilen yayınların veri tabanlarına göre dağılımı sunulmuştur.



Şekil 3.2. Seçilen yayınların veritabanlarına göre dağılımı

3.1.1 Kaos mühendisliği, yazılım sistemlerinin dayanıklılığını artırmak için üretim ortamlarında nasıl etkin bir şekilde uygulanır?

Günümüzde yazılım sistemlerinin karmaşıklığı arttıkça, beklenmeyen hata ve arızalara karşı sistemin dayanıklılığının sürekli olarak test edilmesi kritik bir gereklilik haline gelmiştir. Kaos Mühendisliği, özellikle mikroservis mimarileri için, üretim ortamlarında sistemin dayanıklılığını ölçmek ve artırmak amacıyla uygulanan bilimsel ve sistematik bir yaklaşımdır. SLR'da elde edilen bulgulara göre, kaos mühendisliğinin üretim ortamlarında etkin bir şekilde uygulanması aşağıdaki prensiplere dayanmaktadır:

Gerçek Ortamda Kontrollü Hata Enjeksiyonu : Kaos mühendisliği, canlı (production) ortamda, sistem üzerinde kontrollü ve planlı bir şekilde hata veya arıza senaryoları (ör. servis düşmesi, ağ gecikmesi, veri tabanı erişim hatası vb.) oluşturulmasına dayanır. Bu sayede sistemin beklenmedik koşullara karşı davranışı gerçek kullanıcı trafiği altında gözlemlenir ve ölçülür.

Farklı Seviyelerde ve Tiplerde Hata Simülasyonu : Literatürde; risk tabanlı hata enjeksiyonu, servis seviyesi hata enjeksiyonu ve lineage-driven gibi yöntemlerle, sistemin farklı katmanlarında hata simülasyonu yapılmaktadır. Böylece hem bireysel servislerin hem de tüm ekosistemin dayanıklılığı kapsamlı şekilde test edilir.

Otomasyon ve Merkezi Yönetim : Kaos deneylerinin otomatikleştirilmesi ve merkezi olarak yönetilmesi, özellikle karmaşık ve büyük ölçekli mikroservis ekosistemlerinde deneylerin düzenli, tekrarlanabilir ve izlenebilir olmasını sağlar. Bu amaçla ChaosMesh, ChaosMonkey, Gremlin, Litmus gibi araçlar yaygın olarak kullanılmaktadır.

Dayanıklılık Mekanizmalarının Doğrulanması ve İyileştirilmesi : Kaos Mühendisliği, mevcut hata toleransı ve esneklik mekanizmalarının etkinliğinin doğrulanmasını sağlar. Deneyler sonucunda elde edilen bulgular ile mevcut altyapıda bilinmeyen açıklar ve zayıf noktalar tespit edilir, izleme ve alarm sistemlerinin doğruluğu gözden geçirilir ve gerekli iyileştirmeler yapılır.

Sürekli İyileştirme ve Geri Bildirim Döngüsü : Kaos deneylerinden elde edilen sonuçlar, sistem mimarisi ve operasyonel süreçlerin sürekli iyileştirilmesine olanak tanır. Her bir deney sonrası tespit edilen zayıflıklar giderilerek, sistemin arızalara karşı bağışıklık seviyesi (immunity) artırılır.

Proaktif Risk Yönetimi ve Dayanıklı Sistemler : Kaos mühendisliği, sistemde henüz ortaya çıkmamış olan potansiyel riskleri proaktif olarak keşfetmeyi mümkün kılar. Böylece sistemler, gerçek bir arıza yaşanmadan önce güçlendirilir ve yüksek erişilebilirlik sağlanır.

Üretim ortamlarında kaos mühendisliği uygulamaları, yazılım sistemlerinin dayanıklılığını artırmada en etkin ve gerçekçi yöntemlerden biridir. Canlı ortamda kontrollü hata enjeksiyonu ile hem mevcut dayanıklılık test edilir hem de iyileştirici aksiyonlar alınabilir. Böylece, beklenmeyen arızalar karşısında sistemin sürekliliği ve güvenilirliği güvence altına alınmış olur.

3.1.2 Kaos deneyleri için hangi platformlar kullanıldı?

Kaos mühendisliği deneylerinin uygulanmasında farklı mimarilere ve ihtiyaçlara uygun birçok araç ve platform geliştirilmiştir. Sistematik literatür taramasında (SLR) öne çıkan başlıca platformlar aşağıda özetlenmiştir:

ChaosMonkey: Netflix tarafından geliştirilen, bulut altyapılarında rastgele sunucu veya servisleri devre dışı bırakarak sistemin hata toleransını ölçen, kaos mühendisliğinin ilk ve en çok bilinen araçlarından biridir.

Chaos Mesh: Kubernetes tabanlı ortamlarda çalışan, ağ gecikmesi, pod öldürme, CPU/bellek stresi gibi çok çeşitli arıza senaryolarını kolayca enjekte edebilen, açık kaynaklı ve kapsamlı bir platformdur.

Gremlin: Farklı bulut sağlayıcıları ve altyapılarla uyumlu, endüstriyel ölçekte yaygın olarak kullanılan ve kullanıcı dostu bir arayüze sahip ticari bir kaos mühendisliği aracıdır. Gelişmiş hata senaryoları ve otomasyon desteği sunar.

Litmus: Kubernetes ve konteyner tabanlı ortamlar için geliştirilmiş, otomasyon ve sürekli entegrasyon (CI/CD) süreçlerine entegre edilebilen, açık kaynaklı bir kaos mühendisliği platformudur.

Chaos Toolkit: Farklı bulut, konteyner ve hizmet ortamlarında çalışabilen, JSON/YAML tabanlı senaryo tanımlamaları ile esnek ve genişletilebilir deneyler yapılmasına olanak tanıyan açık kaynaklı bir kaos mühendisliği otomasyon aracıdır. Chaos Toolkit, geniş eklenti ekosistemiyle çok çeşitli altyapılarda kullanılabilir.

Bu platformlar, üretim veya üretime yakın test ortamlarında kontrollü hata senaryoları oluşturup sistemin dayanıklılığını ölçmeyi ve zayıf noktaları proaktif şekilde tespit etmeyi sağlar. Literatürde özellikle ChaosMonkey, Chaos Mesh, Gremlin, Litmus ve Chaos Toolkit araçlarının hem akademik hem endüstriyel uygulamalarda sıkça tercih edildiği görülmektedir.

3.1.3 Kaos mühendisliği, sistem esnekliğini artırmada başarılı bir şekilde uygulanmasını sağlamak için mikro hizmet mimarilerine nasıl etkili bir şekilde uygulanır?

Son yıllarda mikroservis mimarileri, sistem esnekliğinin artırılması amacıyla kaos mühendisliği yaklaşımlarının uygulanmasında yaygın olarak tercih edilmektedir. Sistematik literatür taraması (SLR) bulgularına göre, kaos mühendisliği uygulamalarının mikroservis tabanlı sistemlerde başarılı olabilmesi için aşağıdaki temel ilkeler ve yöntemler ön plana çıkmaktadır:

Steady-State Hipotezi ve Deneysel Tasarım: Kaos mühendisliği deneyleri, sistemin olağan (steady-state) davranışının açıkça tanımlanması ve bu davranışı temsil eden metriklerin belirlenmesiyle başlar. Deneyler sırasında sisteme kontrollü şekilde arıza enjekte edilmekte ve sistemin davranışı bu hipoteze göre ölçülmektedir. Bu sayede, sistemdeki dayanıklılık mekanizmalarının doğrulanması ve iyileştirilmesi mümkün olmaktadır.

Hedefli ve Otomatikleştirilmiş Hata Enjeksiyonu: Mikroservis mimarisinin sunduğu modüler yapı sayesinde, farklı tiplerde hata ve arızalar (ör. servis çökmesi, ağ gecikmesi, kaynak tükenmesi vb.) belirli servisler üzerinde hedefli ve sistematik olarak enjekte edilebilmektedir. Literatürde Chaos Mesh, Chaos Monkey, Gremlin, Litmus, Chaos Toolkit gibi araçların yaygın şekilde kullanıldığı ve otomasyon ile bu sürecin sürekli hale getirildiği vurgulanmaktadır.

Merkezi Gözlemleme ve İzleme: Deneyler sırasında sistemin tüm bileşenlerinden merkezi olarak veri toplanması (ör. response time, hata oranı, throughput) ve distributed tracing teknolojilerinin kullanılması, arızaların sistem üzerindeki etkisinin ayrıntılı biçimde analiz edilmesini sağlamaktadır. Böylece sadece yüzeyde görülen hatalar değil, zincirleme etkiler ve sistemdeki zayıf noktalar da ortaya çıkarılabilmektedir [4].

Sürekli İyileştirme ve Geri Besleme Döngüsü: SLR bulgularına göre, kaos mühendisliği uygulamaları tek seferlik bir test olarak değil, sürekli tekrar edilen ve otomasyon ile CI/CD süreçlerine entegre edilen bir pratik olarak uygulanmalıdır. Deneylerden elde edilen sonuçlar ile sistemdeki eksiklikler giderildikten sonra aynı veya benzer deneyler tekrarlanarak sistemin bağışıklık seviyesi sürekli olarak artırılır [5], [6].

Risk Analizi ve Deney Önceliklendirme: Literatürde, mikroservis tabanlı sistemlerde çok sayıda olası arıza ve risk bulunduğundan, yapılacak deneylerin sistematik bir risk analizi (örn. Failure Mode and Effects Analysis - FMEA, Fault Tree Analysis - FTA) ile önceliklendirilmesi gerektiği vurgulanmaktadır. Böylece en kritik bileşenler ve en olası hata senaryoları üzerinde odaklanmak mümkün olmaktadır.

Sonuç olarak, SLR bulguları kaos mühendisliğinin mikroservis mimarilerinde etkin bir şekilde uygulanmasının, kontrollü ve hedefli deneysel hata enjeksiyonu, merkezi gözlemleme, sürekli iyileştirme ve risk tabanlı önceliklendirme süreçlerinin bütünleşik şekilde yürütülmesine bağlı olduğunu göstermektedir. Bu yaklaşımlar sayesinde, karmaşık ve dinamik mikroservis sistemlerinde hem öngörülen hem de öngörülemeyen hata senaryolarına karşı yüksek seviyede esneklik ve dayanıklılık sağlanabilmektedir [4], [9].

3.1.4 Kaos mühendisliğinin merkezi olarak sağlanması, karmaşık sistemler genelinde Kaos deneylerinin yönetimini ne ölçüde etkili bir şekilde kolaylaştırabilir?

Karmaşık ve dağıtık mikroservis tabanlı sistemlerde kaos deneylerinin merkezi olarak yönetilmesi, deneylerin etkinliğini, ölçeklenebilirliğini ve tekrarlanabilirliğini önemli ölçüde artırmaktadır [14]. Sistematik literatür taramasına göre, kaos mühendisliği uygulamalarında merkezi bir kontrol mekanizmasının bulunması, deneylerin planlanmasından yürütülmesine ve sonuçların analizine kadar olan tüm süreci kolaylaştırmakta ve hızlandırmaktadır. Merkezi yönetim sayesinde, çok sayıda mikroservisin bulunduğu ortamlarda kaos deneylerinin koordinasyonu ve otomasyonu mümkün hale gelmektedir. Bu yaklaşım, deneylerin manuel ve dağınık şekilde yönetilmesinin getirdiği karmaşıklığı azaltmakta, sistem genelinde tutarlı ve standardize edilmiş deneylerin uygulanmasını sağlamaktadır [14]. Ayrıca merkezi kontrol, deneylerin tekrarlanabilirliğini ve farklı ortamlar arasında karşılaştırılabilirliğini artırmakta; elde edilen bulguların, sistem genelinde daha hızlı yaygınlaştırılmasına imkan vermektedir.

Özellikle ölçekli mikroservis mimarilerinde, merkezi kaos mühendisliği platformlarının kullanılması; deneylerin sıraya alınması, eş zamanlı farklı servislerde yürütülmesi, etki alanının (blast radius) kontrollü şekilde sınırlandırılması ve geri

dönüş (rollback) süreçlerinin otomatik olarak devreye alınması açısından büyük avantaj sağlamaktadır. Literatürdeki uygulamalarda, merkezi platformların ayrıca veri toplama, analiz ve raporlama süreçlerini de otomatikleştirerek, karar destek mekanizmalarının güçlendirilmesine katkı sunduğu görülmüştür [14], [20]. Sonuç olarak, kaos mühendisliğinin merkezi olarak sağlanması, mikroservis tabanlı karmaşık sistemlerde deneylerin yönetimini etkin, ölçeklenebilir ve sürdürülebilir kılmakta; sistemsel zayıflıkların proaktif olarak tespit edilip iyileştirilmesine önemli derecede katkı sunmaktadır.

3.1.5 İlgili makalelerde bildirilen zorluklar nelerdir?

Mikroservis mimarilerinde kaos mühendisliği uygulamalarına ilişkin literatürde öne çıkan başlıca zorluklar şunlardır:

Karmaşıklık ve Yönetim Zorluğu : Mikroservislerin çokluğundan ve dinamik yapısından dolayı yönetim, izleme ve hata ayıklama süreçleri oldukça karmaşıktır. Servisler arası bağımlılıklar, hata anında sorunun kaynağını tespit etmeyi güçleştirir.

Etkili Deney Senaryosu Tasarımı: Deneylerde hangi hataların ve hangi kombinasyonların sistemin zayıf noktalarını ortaya çıkaracağı bilinmez. Rastgele deney seçimleri, çoğu zaman gerçek arıza senaryolarını yansıtmaz ve kaynak israfına sebep olabilir. Bu nedenle akıllı ve geçmişe dayalı senaryo seçimleri gereklidir [16].

Gerçekçi Test Ortamı ve Veri Eksikliği: Üretim ortamına benzer test ortamlarının oluşturulması ve yeterli miktarda etiketli (normal/anormal) veri toplanması zordur. Gerçek dünyadaki arıza verilerinin azlığı, anomali tespit yöntemlerinin ve deneylerin doğrulanmasını engeller [13].

Arıza Enjeksiyonu ve Otomasyon Maliyetleri: Arıza enjeksiyonunu otomatikleştirmek, yüksek altyapı ve yazılım entegrasyon maliyetleri gerektirir. Ayrıca, üretim ortamında yapılan deneylerin potansiyel "blast radius"u iyi yönetilmelidir. Yanlış yapılandırılmış deneyler, sistemin büyük bölümünü olumsuz etkileyebilir [21].

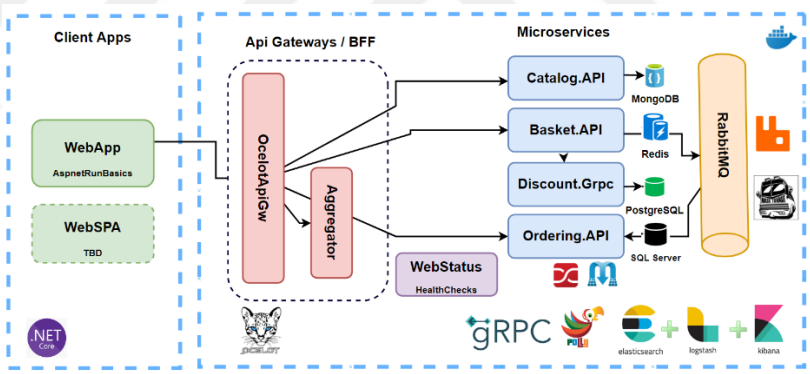
Gözlemlenebilirlik ve İzleme Eksiklikleri: Servislerin poliglot yapısı nedeniyle, yapılan arıza enjeksiyonlarının etkilerini hızlı ve doğru tespit

edebilecek kapsamlı gözlemlenebilirlik ve loglama altyapısına ihtiyaç duyulmaktadır [11].

Tekrarlanabilirlik ve Standart Eksikliği: Deneylerin tekrarlanabilir ve karşılaştırılabilir olması için kullanılan araçlar, deney tanımları ve metriklerin standartlaşması gereklidir. Bu standart eksikliği, farklı sistemlerde elde edilen sonuçların karşılaştırılmasını zorlaştırmaktadır [20].

3.2 Deney Ortamı

Bu çalışmada gerçekleştirilecek olan kaos deneyleri için, açık kaynak kodlu ve mikroservis tabanlı Şekil3.3 de yer alan e-ticaret platformu tercih edilmiştir. Seçilen uygulama, araştırmanın amaçlarına uygun olarak basit, modüler ve yaygın kullanılan bir mimariye sahip olup, deneylerin tekrarlanabilir ve gözlemlenebilir şekilde yürütülmesini sağlamaktadır. Uygulamanın tüm bileşenleri, Minikube üzerinde çalışacak şekilde konteyner tabanlı olarak yapılandırılmış ve dağıtık bir mikroservis ortamı oluşturulmuştur.



Şekil 3.3 Açık Kaynak Kodlu Açık Ticaret Platformu Mimarisi

Deneylerde kaos mühendisliği uygulamaları için Chaos Mesh aracı tercih edilmiştir. Chaos Mesh, Kubernetes ortamlarında çeşitli hata senaryolarını sistematik olarak enjekte etmeye imkan tanıyan, açık kaynaklı ve yaygın olarak kullanılan bir kaos mühendisliği platformudur. Deneylerin tam otomasyonla yürütülebilmesi ve sürekli entegrasyon/sürekli dağıtım (CI/CD) süreçlerine entegre edilebilmesi için Jenkins kullanılmıştır. Böylece, kaos deneyleri otomatik olarak tetiklenebilmekte, deney öncesi ve sonrası sistem davranışları kesintisiz olarak izlenebilmektedir.

Deneysel ortamın tüm bileşenleri ve uygulamalar, 24 GB RAM, 8 çekirdekli CPU ve 500 GB NVMe disk kapasitesine sahip, özel olarak tahsis edilmiş bir VPS üzerinde çalıştırılmıştır. Bu donanım konfigürasyonu, mikroservis uygulamasının gerçekçi bir üretim ortamına benzer şekilde çalışmasını ve kaos deneylerinden elde edilen sonuçların güvenilir ve geçerli olmasını sağlamaktadır. Ayrıca, VPS altyapısında yapılan deneyler, sistem kaynaklarının sınırlarının gözlemlenmesine ve farklı arıza senaryolarında sistemin verdiği tepkilerin detaylı olarak analiz edilmesine olanak vermiştir.

Uygulamanın ve deney ortamının yapısı aşağıda özetlenmiştir:

- Uygulama **:** Açık kaynak kodlu, mikroservis mimarisi ile geliştirilmiş bir e-ticaret platformu.
- Konteyner Orkestrasyonu **:** Minikube üzerinde çalışan Kubernetes ortamı.
- Kaos Deneyi Aracı **:** Chaos Mesh (Mimarisi Şekil 3.4 de sunulmuştur.)
- CI/CD Otomasyonu : Jenkins ile tam otomasyon sağlanmıştır.
- VPS Sunucu : 24 GB RAM, 8 vCPU, 500 GB NVMe SSD diskli bir VPS.

Bu ortam üzerinde gerçekleştirilen kaos deneyleriyle; mikroservislerin farklı hata ve arıza senaryoları karşısındaki dayanıklılıkları sistematik olarak test edilmiş, deneyler tekrarlanabilir ve karşılaştırılabilir bir şekilde yürütülmüştür. Böylece, mikroservis mimarisinde hata toleransı ve esneklik gibi önemli sistem özelliklerinin ölçülmesi ve değerlendirilmesi sağlanmıştır.

3.3 Kullanılan Araç ve Teknolojiler

Bu çalışmada, mikroservis mimarisinde kaos mühendisliği deneylerinin otomatikleştirilmesi ve sistem esnekliğinin ölçülmesi amacıyla bir dizi modern araç ve teknoloji kullanılmıştır. Deney ortamının oluşturulması, deneylerin yönetimi ve performans analizlerinin yapılabilmesi için seçilen araçlar, alan yazında yaygın olarak kullanılan ve ilgili araştırma konularında referans kabul edilen teknolojilerdir.

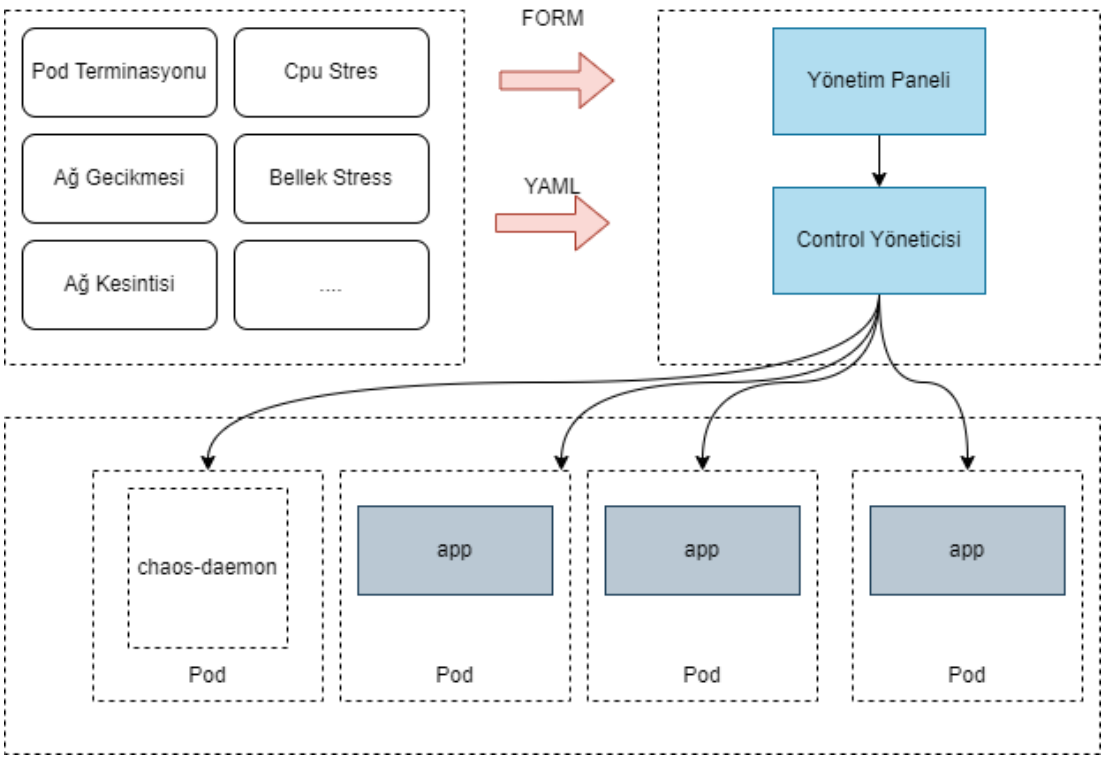
3.3.1 Minikube

Konteyner tabanlı mikroservis mimarisinin kurulumu ve yönetimi için Minikube kullanılmıştır. Minikube, yerel bir Kubernetes kümesi oluşturarak, gerçek

bir bulut ortamına benzer şekilde mikroservislerin dağıtık olarak orkestre edilmesini sağlar. Bu yapı sayesinde mikroservislerin farklı senaryolarda gözlemlenmesi ve deneylerin izole bir ortamda tekrarlanabilir biçimde yürütülmesi mümkün olmuştur.

3.3.2 Chaos Mesh

Kaos mühendisliği deneylerinin uygulanması için açık kaynak kodlu ve bulut yerel sistemlerde yaygın olarak kullanılan Chaos Mesh aracı tercih edilmiştir. Chaos Mesh, Kubernetes tabanlı ortamlarda ağ gecikmesi, pod terminasyonu ve kaynak tüketimi gibi çeşitli hata senaryolarını otomatik ve denetimli bir biçimde enjekte etme imkanı sunmaktadır. Şekil 3.4'te mimari yapısı verilen bu araç, sistemin beklenmedik durumlara karşı gösterdiği tepkilerin sistematik olarak incelenmesine olanak tanımaktadır. Ayrıca, deneylerin otomasyon süreçlerine kolayca entegre edilebilmesi, bu aracı özellikle tercih edilir kılmıştır.



Şekil 3.4 ChaosMesh Mimari Yapısı

3.3.3 Jenkins

Sürekli entegrasyon ve sürekli dağıtım (CI/CD) süreçlerinin yönetimi ve deneylerin tam otomasyon ile yürütülmesi için Jenkins kullanılmıştır. Jenkins, deneylerin başlatılması, uygulamanın yeniden dağıtılması ve deney sonuçlarının toplanması süreçlerinin otomatikleştirilmesini sağlamıştır. Böylece, deneylerin insan müdahalesi olmadan, tutarlı ve tekrarlanabilir şekilde uygulanması mümkün olmuştur.

3.3.4 Apache JMeter

Deney sürecinde, mikroservislerin farklı hata koşulları altındaki performansını ölçmek ve karşılaştırmak amacıyla Apache JMeter aracı kullanılmıştır. JMeter, dağıtık sistemlerde yük testi ve performans analizi yapmak için yaygın şekilde kullanılan bir açık kaynak aracıdır. Bu çalışmada, deneyler sırasında sistemin yanıt süreleri ve throughput gibi metrikleri izlenmiş ve elde edilen veriler istatistiksel olarak değerlendirilmiştir. Böylece, kaos deneylerinin sistem üzerindeki etkisi nesnel ve ölçülebilir şekilde analiz edilmiştir.

3.3.5 Diğer Bileşenler ve Altyapı:

Tüm uygulama ve deneyler, 24 GB RAM, 8 çekirdekli CPU ve 500 GB NVMe SSD'ye sahip bir VPS üzerinde gerçekleştirilmiştir. Bu kaynaklar, hem mikroservislerin hem de deneylerin gerektirdiği işlem gücünü ve depolama kapasitesini sağlayarak, deney ortamının gerçekçi bir üretim sistemine yakın koşullarda çalışmasını mümkün kılmıştır. Kullanılan araçlar ve teknolojiler ile oluşturulan deney ortamı, mikroservis mimarisinde hata toleransı, dayanıklılık ve otomasyonun bütüncül olarak değerlendirilmesini sağlamış; elde edilen veriler, Kaos Mühendisliği uygulamalarının pratikteki etkilerinin derinlemesine incelenmesine olanak tanımıştır.

3.4 Kaos Deneylerinin Tasarlanması

Bir yazılım sisteminin beklenmedik durumlar karşısında nasıl bir tepki verdiğini anlamak için yapılan çalışmalar, öncelikle sistemin olağan halinde nasıl davrandığını gözlemleyerek başlar. Bu gözlemden sonra, hangi tip arızaların ya da aksaklıkların test edileceği seçilir. Genellikle ağda gecikmeler oluşması, bir hizmetin aniden durması veya kaynakların yoğun şekilde kullanılması gibi durumlar ele alınır.

Belirlenen sorun kontrollü bir şekilde sistem üzerinde uygulanır. Süreç boyunca çeşitli performans göstergeleri yakından takip edilir ve test bitiminde bu göstergelerdeki değişiklikler, başlangıçtaki ölçümlerle kıyaslanır. Böylelikle sistemin hangi tür olumsuzluklara karşı ne tür zafiyetler gösterdiği anlaşılır. Bu tür deneylerin sürekli ve düzenli biçimde yürütülebilmesi adına otomatikleştirilmiş test ve izleme araçlarından da destek alınır. Sonuç olarak, mikroservislerle kurulu sistemlerin alışılmadık hatalara karşı dirençleri, sistematik bir yaklaşımla gözlemlenmiş olur.

3.4.1 Pod Terminasyonu

Sistem üzerinde dayanıklılığın ve olası beklenmedik arızalara karşı gösterilen reaksiyonların anlaşılması amacıyla gerçekleştirilen çalışmalardan biri de, çalışan birimlerden birinin aniden devre dışı bırakılmasıdır. Bu yöntemde, hizmeti oluşturan kubernetes pod'larından biri rastgele seçilip beklenmedik bir şekilde sonlandırılır. Böylece, sistemin bu ani kayba karşı nasıl davrandığı, ne tür mekanizmaların devreye girdiği ve hizmetin sürekliliği üzerindeki etkileri değerlendirilir. Uygulamanın aksamadan çalışmaya devam edip etmediği, yeniden yapılanma süreçlerinin hızı ve genel işleyişe yansıyan sonuçlar gözlemlenir. Bu sayede, gerçek bir problem ortaya çıkmadan önce sistemin toparlanma kapasitesi ve dayanıklılık düzeyi hakkında önemli bilgiler elde edilir.

3.4.2 Ağ Gecikmesi

Sistemler üzerinde yapılan bazı deneylerde, hizmetler arası veri iletiminde bilinçli olarak gecikme yaratmak, sistemin bu tür olumsuzluklara nasıl karşılık verdiğini görmek için kullanılır. Ağ gecikmesi uygulandığında, bir servisin diğerine gönderdiği mesajlar normalden daha yavaş iletilir. Bu tür bir durum, gerçek hayatta beklenmeyen trafik artışları, fiziksel altyapıdaki sorunlar veya bölgesel yavaşlamalar gibi pek çok nedenle yaşanabilir. Deneyin amacı, sistemde oluşan aksamanın ne kadar sürede fark edildiğini, hizmet kalitesinin bu durumdan nasıl etkilendiğini ve uygulamanın olağan akışına dönüş sürecini gözlemlemektir. Böylece, arka planda çalışan hata yönetimi ya da yük dengeleme gibi mekanizmaların etkinliği daha net ortaya konur.

Pratikte, mikroservis tabanlı bir yapıda bu tarz gecikmelerin servisler arasındaki işleyişi yavaşlatıp yavaşlatmadığı ya da zincirleme bir etki yaratıp yaratmadığı gözlenir. Buradaki esas mesele, sadece hata anında verilen tepki değil,

sistemin bu gibi koşullarda dengesini koruyup koruyamadığıdır. Kimi zaman ufak bir gecikme, çok daha büyük bir sorunun öncüsü olabilir. Deneyde bu tür sinyallerin tespiti büyük önem taşır. Sonuç olarak, ağ gecikmesi deneyiyle elde edilen bulgular, gerçek dünyadaki dalgalanmalara karşı sistemin ne derece dayanıklı olduğunu ortaya çıkarır.

3.4.3 Ağ Kesintisi

Bazı senaryolarda, hizmetler arasındaki iletişimin tamamen kopması ya da belirli bir süre erişilemez hale gelmesi gündeme gelebilir. Ağ kesintisi deneyi, bu tarz bir kopukluğun sistem üzerindeki etkilerini incelemeye olanak tanır. Buradaki yaklaşımda, belirli mikroservisler arasındaki bağlantı bilinçli olarak kesilir ve sistemin bu durumda ne tür tepkiler verdiği gözlemlenir. Gerçek dünyada bu tip kesintiler; donanım arızaları, ağ ekipmanındaki beklenmeyen problemler veya servis sağlayıcı kaynaklı bölgesel sorunlar nedeniyle ortaya çıkabilir.

Deney sırasında, uygulamanın kullanıcıya hizmet sunmaya devam edip edemediği, hatalı isteklerin nasıl ele alındığı ve sistemin otomatik iyileşme mekanizmalarının devreye girip girmediği gibi konular özellikle izlenir. Ayrıca, kesinti süresinin uzaması halinde meydana gelebilecek zincirleme hataların ve veri tutarsızlıklarının önüne geçilmesi için sistemin hangi yolları tercih ettiği anlaşılmaya çalışılır. Sonuç olarak, ağ kesintisi deneyiyle, uygulamanın tamamen beklenmedik bir kopukluğa karşı dayanıklılığı gerçekçi bir şekilde test edilmiş olur. Bu sayede, sistemin görünmeyen zayıf noktaları önceden keşfedilerek gerekli iyileştirmeler yapılabilir.

3.4.4 Cpu ve Bellek Stres

Bir sistemin sınırlarını görmek için başvurulan yaygın yöntemlerden biri de işlemci (CPU) ve bellek (RAM) kaynaklarını zorlayacak şekilde yapılan stres testleridir. Bu tip deneylerde, belirli mikroservisler üzerinde olağan dışı derecede işlem gücü veya bellek tüketimi oluşturularak, altyapının bu aşırı yüke nasıl yanıt verdiği incelenir. Gerçek hayatta ise bu tarz yoğunluk; beklenmeyen trafik patlamaları, yazılım hataları ya da plansız işlemler sonucu aniden ortaya çıkabilir.

Deneyin temel amacı, sistemde hizmetlerin yavaşlaması, hata mesajları veya hizmet reddi gibi tepkilerin zamanlamasını ve şiddetini gözlemlemektir. Ayrıca,

kaynakların tamamen tükenmesi halinde sistemin kendi kendini toparlama becerisi, yük dengeleme stratejilerinin etkinliği ve hata yönetiminin başarısı ön plana çıkar. Stres testinin en önemli yönü, yazılımın olağan sınırlarının ötesinde bir duruma sürüklenmesiyle birlikte, zayıf halkaların ve performans darboğazlarının ortaya çıkmasını sağlamasıdır. Böylece, sistemin gerçek bir kriz anında nasıl davrandığı ve hangi noktada müdahale gerektirdiği anlaşılır. Elde edilen bulgular, sadece teknik iyileştirmeler için değil, aynı zamanda operasyonel hazırlık açısından da yol gösterici olur.

3.5 Veri Toplama ve Analiz Yöntemleri

Bu tez çalışmasında, mikroservis mimarisinde kaos deneylerinin otomatikleştirilmiş bir şekilde yürütülmesi kapsamında, veri toplama ve analiz süreçleri çok katmanlı ve bütüncül bir yaklaşımla ele alınmıştır. Deneysel süreç boyunca, sistem üzerinde gerçekleştirilen hata enjeksiyonlarının etkilerini ortaya koymak amacıyla, hem altyapı hem de uygulama düzeyinde çeşitli izleme ve ölçüm mekanizmaları kullanılmıştır [14][24]. Her bir deney tipi için öncelikle sistemin olağan çalışma koşullarındaki ("steady-state") davranışı tanımlanmış, bu davranışa ilişkin temel metrikler sürekli trafik üreten bir yük testi (ör. JMeter scriptleri) yardımıyla toplanmıştır [24]. Kaos deneyleri sırasında, seçili mikroservislere kontrollü bir şekilde hata enjekte edilerek, bu müdahalelerin sistemin yanıt süresi, hata oranı, kaynak kullanımı ve hizmet sürekliliği gibi temel performans göstergeleri üzerindeki etkileri merkezi bir loglama altyapısı üzerinden kaydedilmiştir [10].

Veri toplama sürecinde Python tabanlı otomasyon betikleri devreye alınmış, deneylerin farklı fazlarında (deney öncesi, deney anı, iyileşme süreci) oluşan ham veriler (JSON, CSV, log) birleştirilip ön işleme tabi tutulmuştur. Böylece, deneylerin tekrarlanabilirliği ve karşılaştırılabilirliği için gerekli veri bütünlüğü sağlanmıştır. Özellikle, deneylerin istatistiksel olarak anlamlı şekilde analiz edilmesi için, deney ve kontrol grupları arasındaki farklılıklar Kolmogorov-Smirnov gibi parametrik olmayan testlerle değerlendirilmiştir. Otomatik analiz betikleri ile, metriklerin fazlara göre ayrıştırılması, anomali tespiti ve görsel raporların oluşturulması sağlanmıştır. Sonuç olarak, bu çok katmanlı ve otomasyon odaklı veri toplama-analiz yöntemiyle; mikroservis temelli sistemlerde kaos mühendisliğinin etkinliği nesnel ve tekrar edilebilir biçimde ortaya konmuştur. Yöntemin güçlü yönü, hem literatürde önerilen

iyi uygulamalara hem de gerçek ortamda pratik gerekliliklere yanıt verecek şekilde kurgulanmış olmasıdır.

4. DENEYLERİN SONUÇLARI

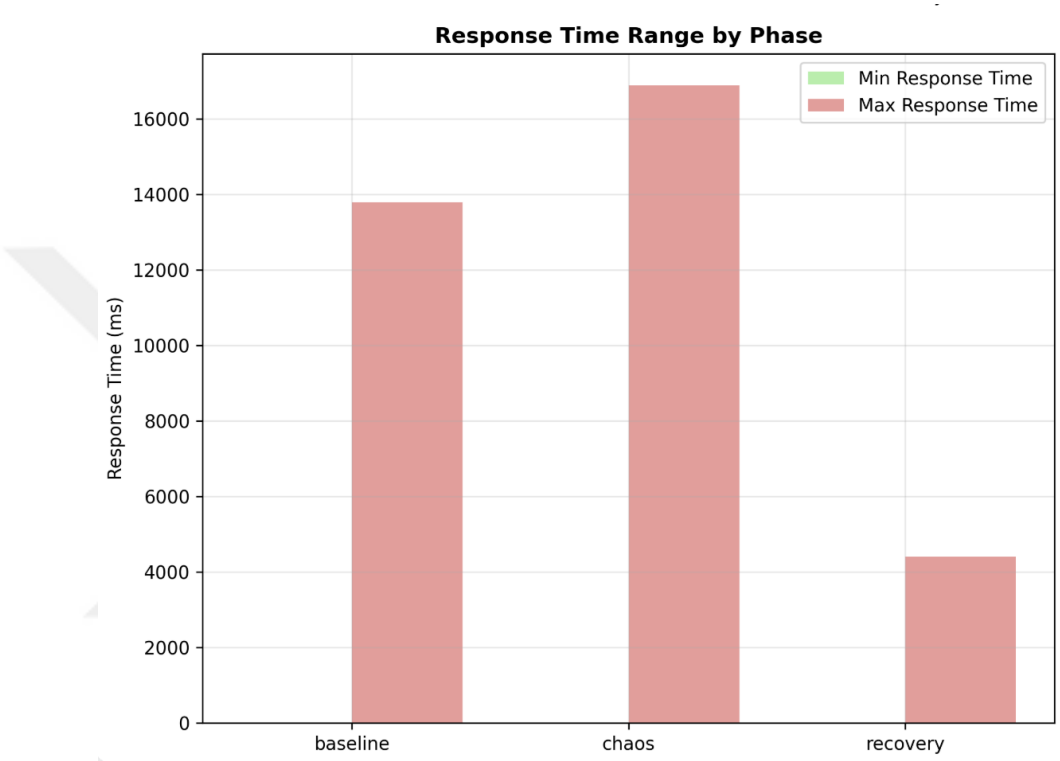
Bu tez kapsamında gerçekleştirilen kaos mühendisliği deneyleri, mikroservis mimarisine sahip sistemlerin beklenmedik hata durumlarına karşı dayanıklılık ve toparlanma kapasitelerini ölçmek amacıyla tasarlanmıştır. Kaos mühendisliği, canlı sistemlerde kontrollü hata enjekte ederek sistemin hata toleransı ve hata yönetim mekanizmalarının etkinliğini değerlendiren deneysel bir yaklaşımdır. Literatürde belirtildiği üzere, mikroservis tabanlı uygulamalar karmaşık bağımlılıklara ve dinamik çalışma koşullarına sahip olduğu için geleneksel test yöntemleriyle olası zayıf noktaların tespiti zordur [10][14]. Bu bağlamda, pod terminasyonu, ağ gecikmesi ve ağ kesintisi gibi çeşitli hata senaryolarının canlı ortamda simüle edilmesi, sistem davranışının gerçekçi ve bütüncül biçimde gözlemlenmesine olanak tanımıştır [20]. Elde edilen sonuçlar, hem sistem bileşenlerinin birbirleri üzerindeki etkilerini hem de Kubernetes'in otomatik iyileştirme mekanizmalarının başarısını ortaya koymakla kalmamış, aynı zamanda farklı hata tiplerine göre sistemin performans ve hata oranlarındaki değişimler net bir şekilde analiz edilmiştir. Böylece, kaos deneyleri, sistemin zayıf noktalarının belirlenmesi ve dayanıklılığın artırılması için veri destekli kararların alınmasına katkı sağlamıştır.

4.1 Pod Terminasyon Deney Sonuçları

Mikroservis mimarilerinde pod terminasyonu (pod kill), sistemin hata toleransı ve otomatik iyileşme yeteneklerinin sınanması açısından kritik bir deney türüdür. Bu çalışmada, pod terminasyonu deneyi, basket-api adlı servis üzerinde gerçekleştirilmiş ve sistemin bu ani kayıptan nasıl etkilendiği detaylı olarak gözlemlenmiştir. Deney sürecinde, sistemin steady-state (kararlı durum) hipotezi olarak "basket-api servisinin uç noktalarının belirlenen tepki süresi aralığında kalmaya devam etmesi" kabul edilmiştir. Deneyin ilk aşamasında, servisin yanıt süreleri ve hata oranları normal sınırlar içinde seyrederken, pod'un ani olarak sonlandırılmasıyla birlikte servis, kısa süreliğine erişilemez hale gelmiştir. Bu sürede, hata oranlarında ani bir artış gözlenmiş, response time (yanıt süresi) metriklerinde belirgin bir yükselme meydana gelmiştir.

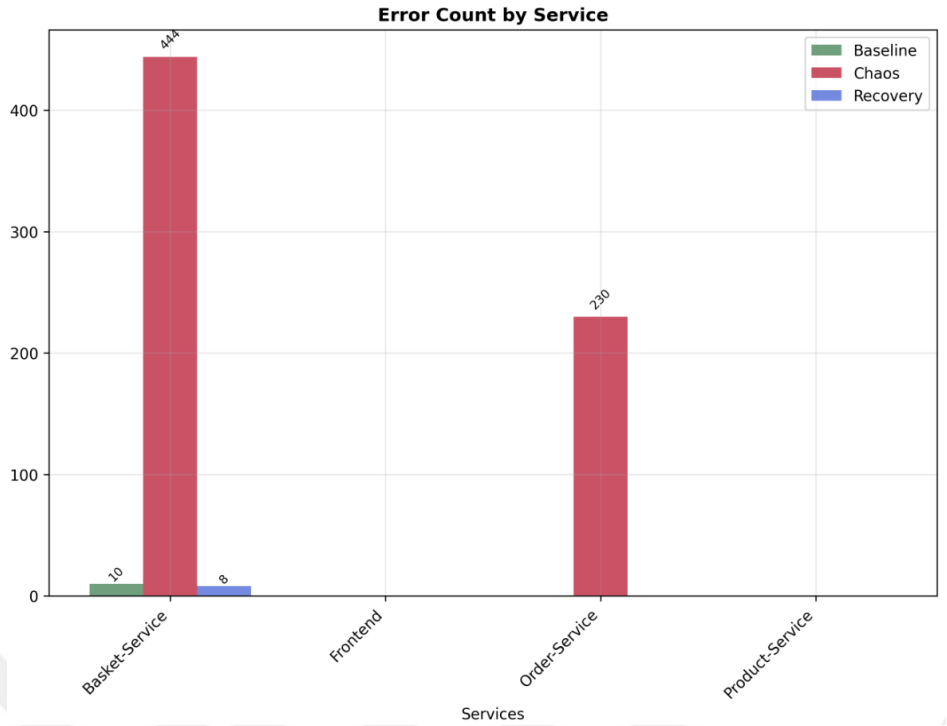
Aşağıdaki grafikte görüldüğü üzere, pod terminasyonu sonrası response time değerlerinde kısa süreli bir zirve oluşmuş ve ardından sistem Kubernetes'in otomatik yeniden başlatma (self-healing) mekanizması sayesinde hızla toparlanmıştır.

Özellikle, sistemdeki trafiğin bir kısmı kısa bir süre 5xx hata kodları ile sonuçlanmış, ancak yeni pod'un devreye alınmasıyla birlikte hatalar normale dönmüştür. Pod terminasyonu öncesi ve sonrası sistem genelinde yapılan isteklerin yanıt sürelerinde yaşanan değişim bu Şekil 4.1 de açıkça izlenmektedir. Ani pod kaybının hemen ardından yükselen yanıt süreleri, otomatik iyileşme süresinde tekrar kararlı bir seviyeye inmiştir.



Şekil 4.1 Pod Terminasyon Sistem Cevap Süreleri

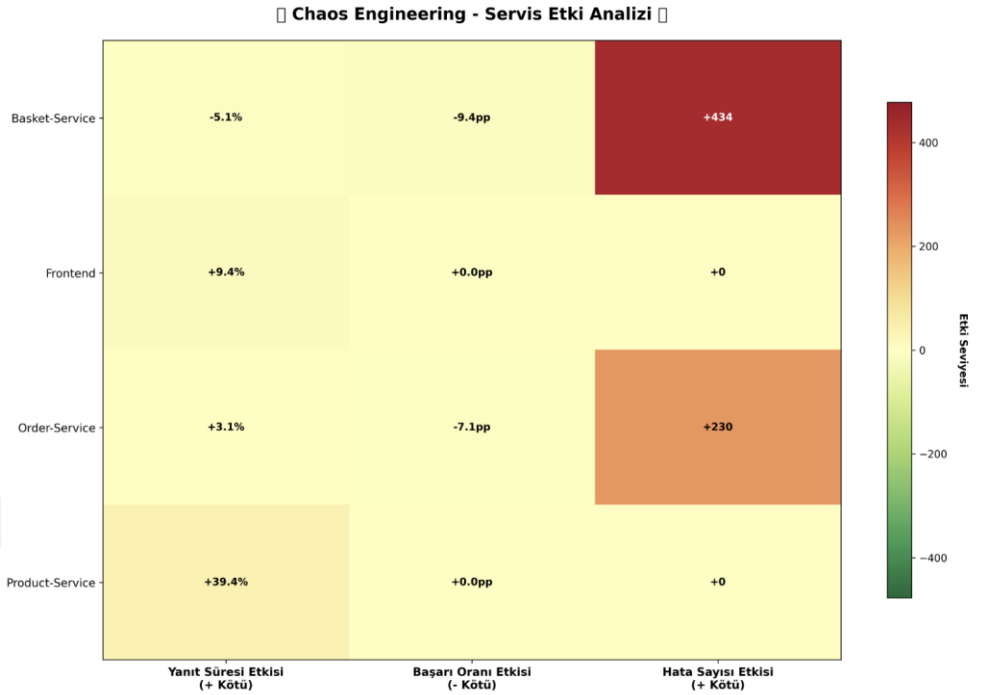
Deney sırasında gözlemlenen 5xx hata oranı Şekil 4.2 de, pod'un terminasyonuyla başlayan hata artışı, kubernetesin yeni pod'u başlatması ve yük dengelemenin sağlanmasıyla birlikte hızlıca düşüş göstermiştir.



Şekil 4.2 Pod Terminasyon Servis Hata Sayıları

Pod terminasyonu deneyi sonucunda Şekil 4.3 te de gösterildiği üzere oluşan Chaos Etki Haritası, sistemdeki etkilerin zamana yayılımını ve farklı servislere olan yansımalarını görsel olarak ortaya koymaktadır. Bu harita sayesinde, özellikle pod

kaybı sonrası oluşan ani hata artışı ve sistemin kısa süre içinde toparlanma süreci bütüncül bir şekilde izlenebilmektedir.



Şekil 4.3 Pod Terminasyon Servis Etki Analizi

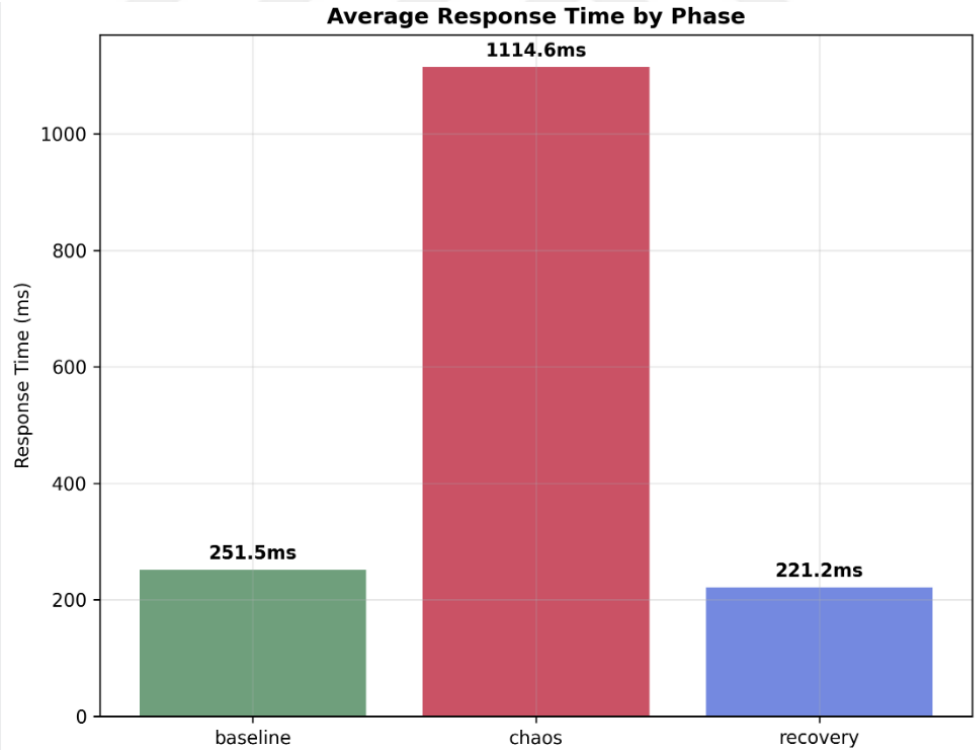
Bu deneyin sonuçları, mikroservis mimarisinde sağlanan otomatik iyileşme yeteneğinin pratikte başarılı bir şekilde çalıştığını göstermektedir. Sistem, pod terminasyonuna rağmen kısa sürede toparlanmış ve kullanıcıya minimum düzeyde kesinti yaşatmıştır. Bununla birlikte, yük altında yapılan deneylerde yanıt sürelerinde ve hata oranlarında gözlenen geçici artışlar, kritik hizmetlerin yüksek erişilebilirlik gereksinimleri için ek iyileştirme stratejilerinin (örneğin, pod sayısının artırılması veya pod dağılımının daha dengeli yapılması gibi) dikkate alınması gerektiğini ortaya koymaktadır.

Sonuç olarak, pod terminasyonu gibi ani hata senaryoları, sistemin dayanıklılığını ölçmek ve Kubernetes'in otomatik kurtarma süreçlerinin etkinliğini gözlemlemek için oldukça değerli veriler sağlamaktadır. Bu tür deneyler, gerçek canlı ortamlarda olası arızalara karşı hazırlıklı olmak ve sistem tasarımında iyileştirmeler yapmak açısından kritik önemlidir.

4.2 Ağ Gecikmesi Deney Sonuçları

Bu bölümde, mikroservis mimarisinde uygulanan ağ gecikmesi deneyi ile elde edilen gözlemler ve sonuçlar aktarılacaktır. Deneyde, basket-api isimli servisin ağ katmanına suni gecikme enjekte edilmiş ve sistemin bu değişken koşullar altındaki davranışı izlenmiştir. Deney öncesinde sistem kararlı çalışırken, gecikme enjekte edildiği anda servislerin yanıt sürelerinde Şekil 4.4 te de olduğu gibi dikkat çekici bir artış meydana gelmiştir. Özellikle ortalama yanıt süresinin 251ms'den 1115ms'ye çıkması ve P95 değerinin 4.5 saniyeyi aşması, sistemin bu tür beklenmeyen gecikmelere karşı duyarlılığını ortaya koymaktadır. Ağ gecikmesi boyunca hata oranında belirgin bir yükseliş izlenmemiştir, ancak toplam istek sayısı düşmüş ve bazı istekler gecikmeler sebebiyle tamamlanamamıştır. Bu bulgular, yüksek trafik altında kullanıcı deneyiminin olumsuz yönde etkilenebileceğine işaret etmektedir.

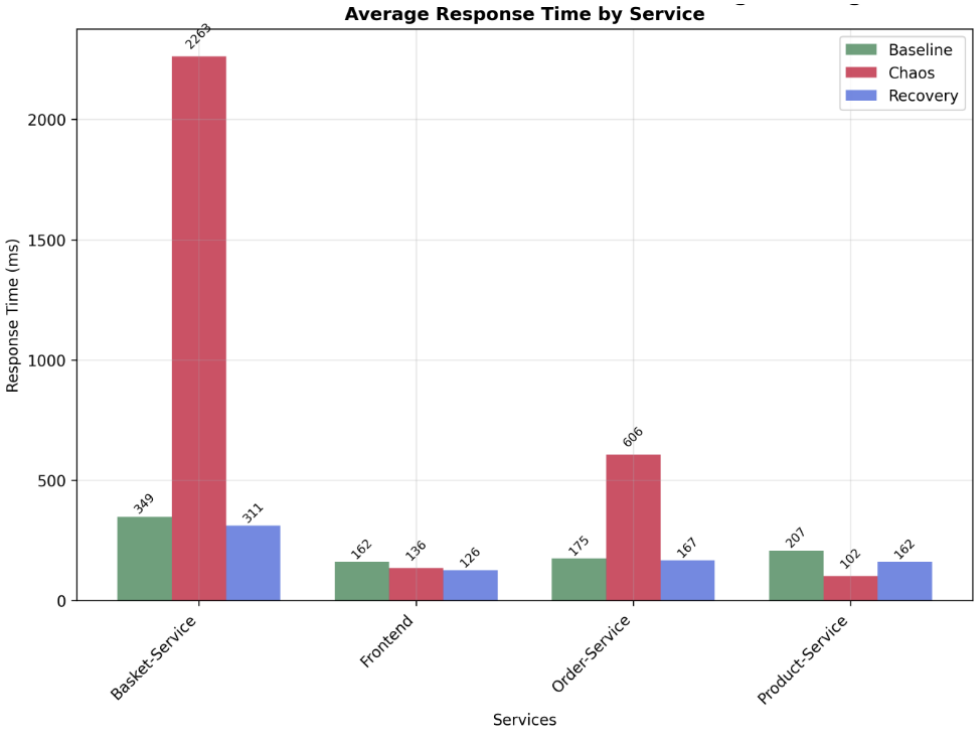
Aşağıda, ağ gecikmesi deneyinde ölçülen yanıt sürelerinin fazlara göre değişimini gösteren grafik yer almaktadır.



Şekil 4.4 Ağ Gecikmesi Sistem Ortalama Cevap Süreleri

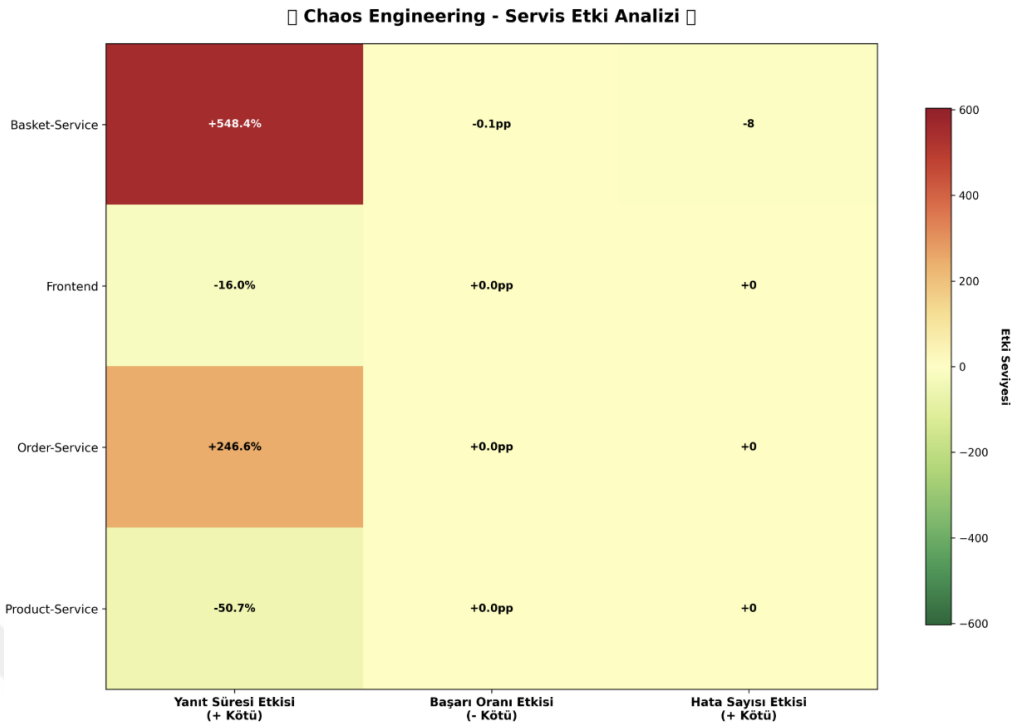
Deney süresince yalnızca basket-api değil, ona bağlı çalışan diğer servislerde de benzer bir performans kaybı gözlenmiştir. Özellikle basket-service ve order-service, yanıt sürelerindeki yüksek artış ile sistemin en zayıf halkaları olarak öne çıkmıştır. Sistemin toparlanma fazında ise, gecikme etkisi ortadan kaldırıldığında ortalama yanıt süresi 221ms'ye gerilemiş ve servisler büyük ölçüde eski kararlı hallerine dönmüştür.

Cevap sürelerinin servis bazında deney süresince izlenimi Şekil 4.5'te gözlemlenmektedir.



Şekil 4.5 Ağ Gecikmesi Servis Bazında Ortalama Cevap Süreleri

Ağ gecikmesi etkisi sistemden kaldırıldıktan sonra, otomatik iyileşme mekanizmalarının devreye girerek trafiği hızlıca normal seviyelere taşıdığı gözlemlenmiştir. Bununla birlikte, kritik servislerdeki yanıt süresi artışları, uygulama mimarisinde darboğaz oluşturabilecek noktaların varlığını da açığa çıkarmaktadır. Deneyin bir parçası olarak hazırlanan Chaos Etki Haritası, sistem genelinde gecikme etkisinin nasıl yayıldığını ve hangi bileşenlerin daha fazla etkilendiğini bütüncül olarak Şekil4.6da da göstermektedir. Özellikle gecikmeye hassas olan servisler etki haritası üzerinde daha belirgin şekilde öne çıkmakta, bu da olası bir performans sorununun sistem genelinde zincirleme etkiler yaratabileceğini kanıtlamaktadır.

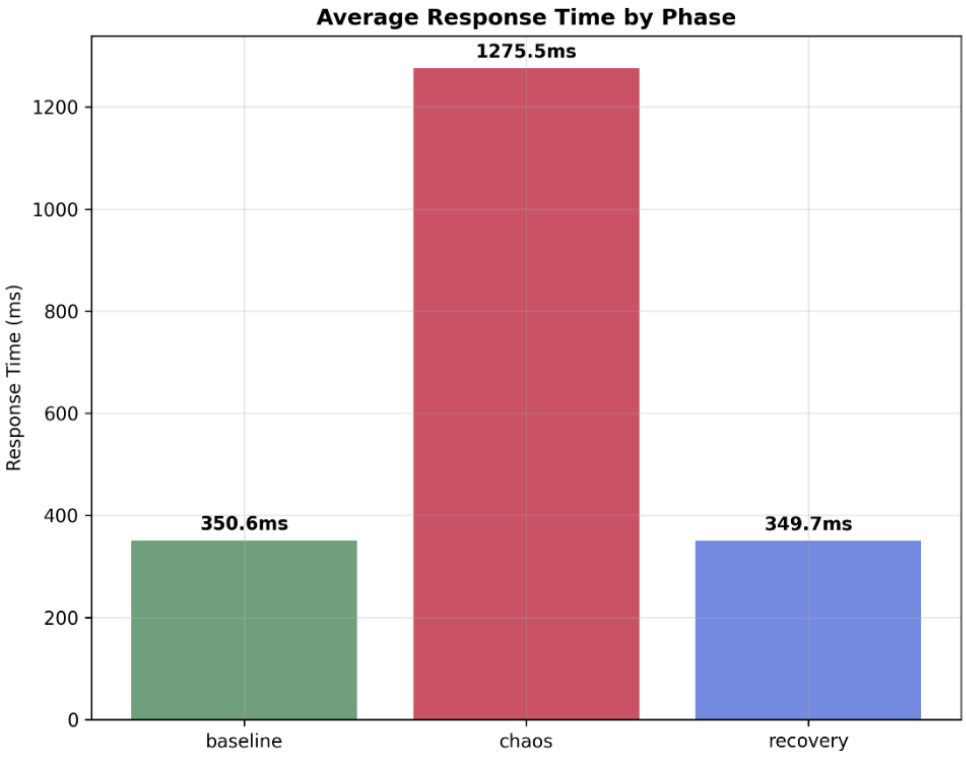


Şekil 4.6 Ağ Gecikmesi Servis Etki Analizi

Bu deneyden elde edilen sonuçlar, mikroservis mimarisinde ağ gecikmesinin sadece hedeflenen servisi değil, tüm bağlı bileşenleri etkileyebileceğini; sistemde devre kesici ve zaman aşımı gibi dayanıklılık mekanizmalarının hayati olduğunu ve iyileştirme gereksinimlerinin belirginleştiğini göstermektedir. Sonuç olarak, ağ gecikmesi senaryosu, mimari tasarımda proaktif önlemlerin ve sürekli gözlemin gerekliliğine dikkat çekmektedir.

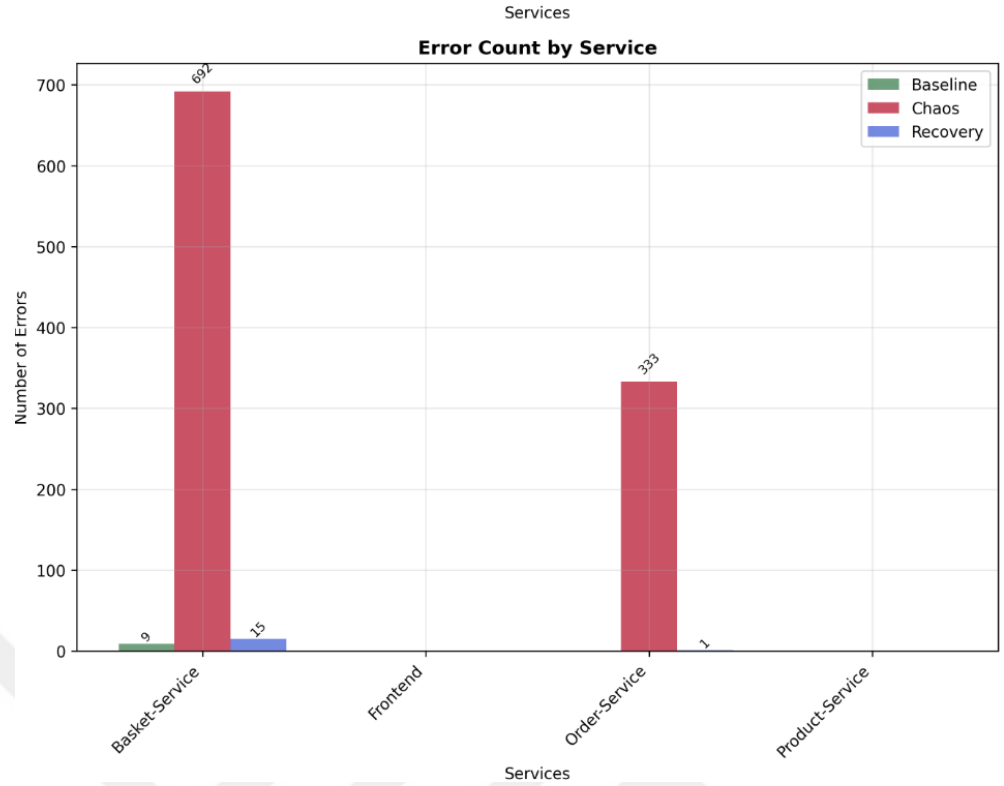
4.3 Ağ Kesintisi Deney Sonuçları

Ağ kesintisi deneyi sırasında sistemin davranışı detaylı biçimde gözlemlenmiştir. Deney boyunca belirli aralıklarla uygulanan paket kaybı (network loss) enjeksiyonları, özellikle mikroservisler arasındaki iletişimi doğrudan etkilemiştir. Deneyin başlangıcında sistemin kararlı bir şekilde çalıştığı gözlemlenirken, ağda kayıp oranı artırıldıkça yanıt sürelerinde belirgin dalgalanmalar ve hata oranında artış meydana gelmiştir.



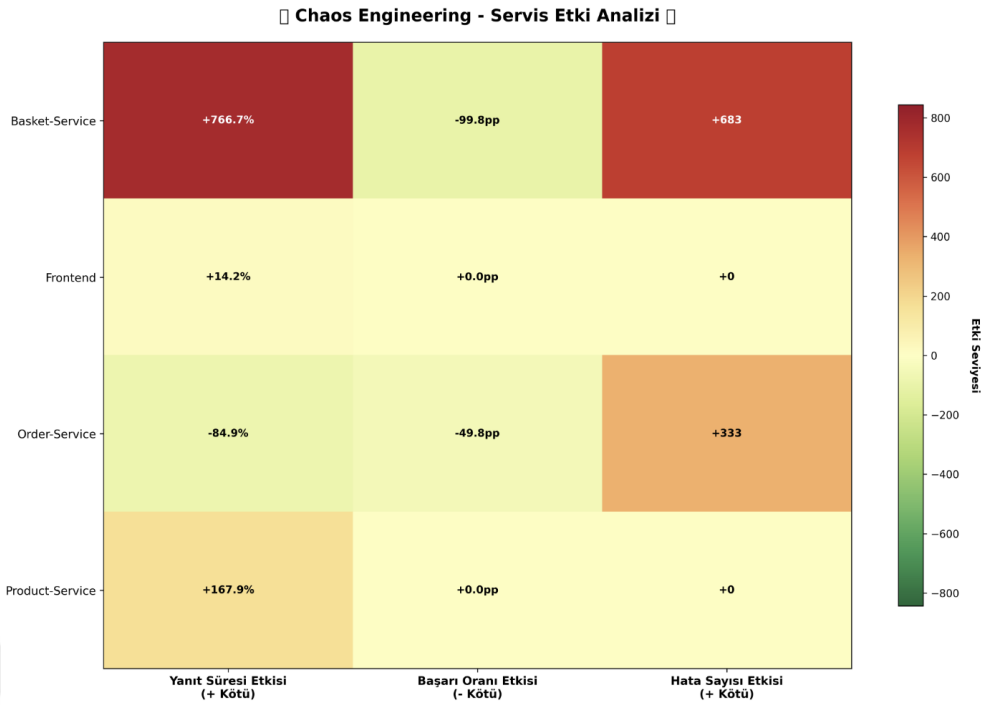
Şekil 4.7 Ağ Kesintisi Sistem Ortalama Cevap Süreleri

Burada görülen ilk grafik, ağ kaybı uygulandığında sistemdeki ortalama yanıt süresinin önemli ölçüde yükseldiğini Şekil4.7 ortaya koymaktadır. Özellikle %10 ve üzeri paket kaybı seviyelerinde gecikmelerin belirginleştiği, zaman zaman kritik eşiklerin de aşıldığı tespit edilmiştir. Aynı deney boyunca hata oranlarında da dikkat çekici bir artış Şekil4.8 üzerinde de görüldüğü üzere tespit edilmiştir. Sağlıklı çalışmada neredeyse sıfıra yakın seyreden hata oranı, ağ kaybı ile birlikte keskin bir biçimde yükselmiştir.



Şekil 4.8 Ağ Kesintisi Servis Bazlı Hata Sayıları

Burada yer alan ikinci grafik, sistemde oluşan hata oranlarının, özellikle belirli ağ kaybı oranlarında ani sıçramalarla arttığını göstermektedir. Bu, sistemdeki bazı servislerin veya bağımlılıkların, ağ kesintisine karşı daha hassas olduğunu işaret etmektedir. Ağ kesintisinin etkisiyle oluşan zincirleme etkileri ve sistemdeki kritik bağımlılık noktalarını görselleştirmek amacıyla Chaos Etki Haritası da kullanılmıştır. Chaos Etki Haritası incelendiğinde, ağ kaybının özellikle belirli servisler üzerinde daha fazla baskı yarattığı ve sistemin dayanıklılığını bu noktalarda zorladığı net biçimde Şekil 4.9 üzerinde de görülmektedir. Bağımlılıkların ve veri akışının yoğun olduğu alanlarda, en küçük bir kesintinin bile genel sistem davranışını ciddi biçimde etkileyebileceği ortaya çıkmıştır.

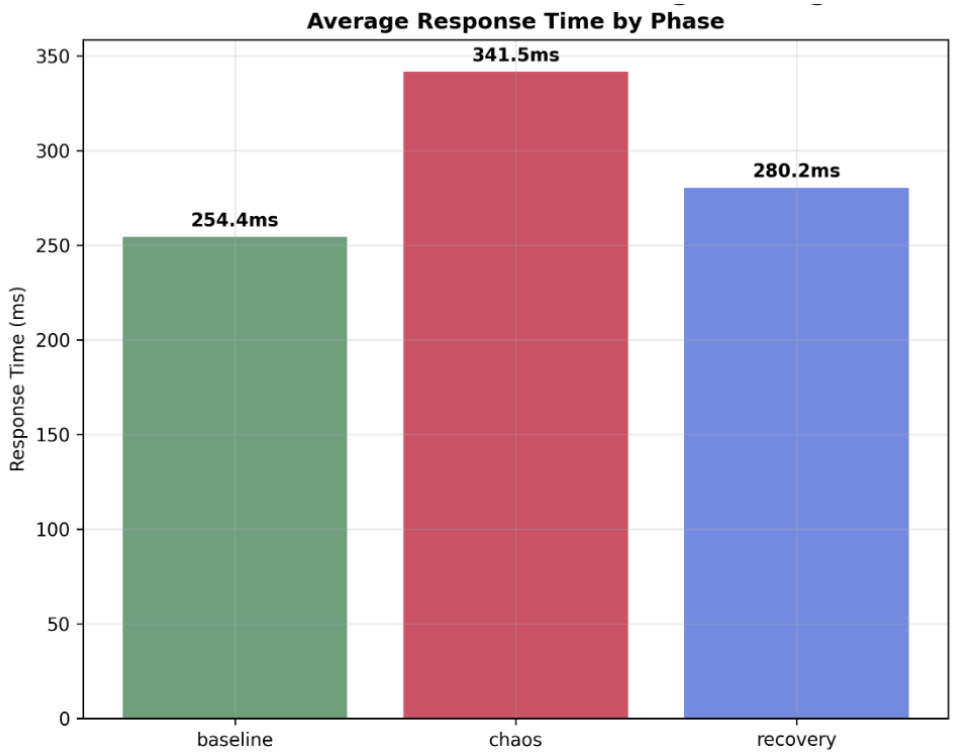


Şekil 4.9 Ağ Kesintisi Servis Etki Analizi

Genel olarak, deneyin bulguları; mikroservis tabanlı sistemlerin gerçek hayattaki ağ sorunlarına karşı hazırlıklı olması gerektiğini, ağ seviyesindeki kayıpların göz ardı edilmesinin ciddi hizmet kesintilerine yol açabileceğini göstermektedir. Deney sonunda elde edilen grafikler ve etki haritası, sistemin zayıf noktalarını açıkça ortaya koymuş ve mimaride alınabilecek önlemler için önemli bir referans sunmuştur.

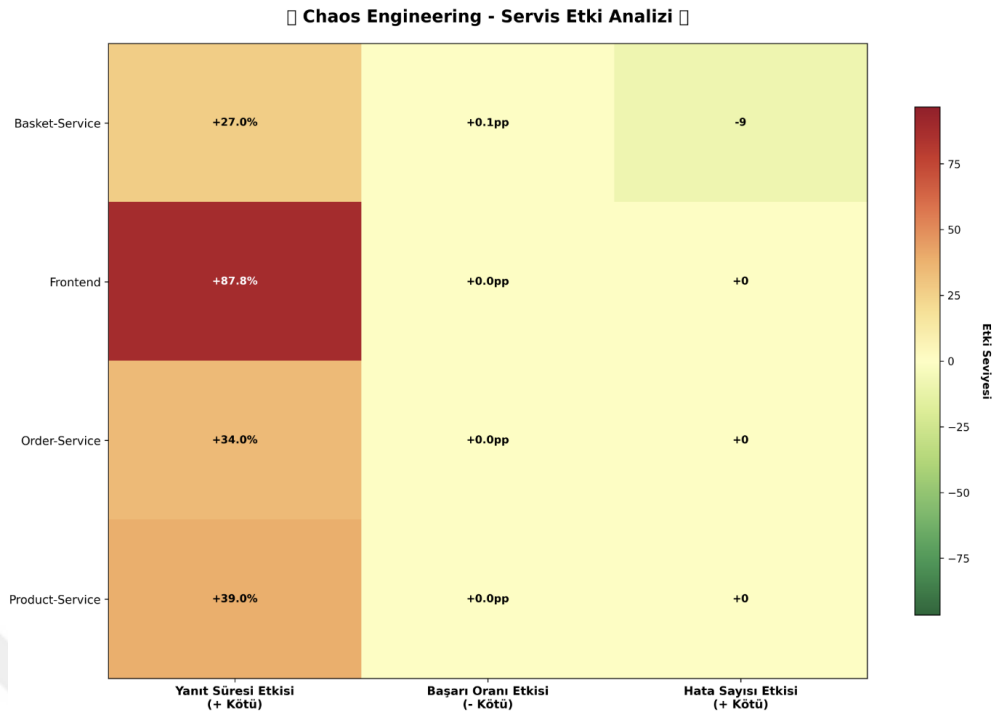
4.4 Cpu Stres Deney Sonuçları

Yapılan CPU stres deneyi süresince, sistemin temel davranışlarında özellikle yanıt süresi üzerinde Şekil 4.10 da gösterildiği üzere dikkat çekici değişiklikler kaydedilmiştir. Deneyin başlangıcında servislerin verdiği yanıt süreleri belirli bir istikrar gösterirken, CPU üzerindeki yük arttıkça yanıt sürelerinde gözle görülür bir yükselme ortaya çıkmıştır. Özellikle stres uygulamasının zirveye ulaştığı anlarda, servis gecikmeleri belirgin şekilde artmıştır.



Şekil 4.10 Cpu Stres Sistem Ortalama Cevap Süreleri

Hata oranı açısından ise, deney boyunca sistemde ciddi bir hata birikimi gözlenmemiştir. Deney sonunda elde edilen veriler, sistemin CPU kaynakları zorlandığında dahi büyük oranda yanıt verebildiğini göstermektedir. Ancak, artan gecikmeler kullanıcı deneyimi ve uygulama akışı açısından önemli bir risk faktörü olarak ortaya çıkmıştır. Deneyin sistem genelindeki etkisi ise Şekil 4.11 deki Chaos Etki Haritası ile analiz edilmiştir. Diyagram üzerinde, CPU stresinin başlatıldığı andan itibaren hem hedef serviste hem de ilişkili bileşenlerde oluşan dolaylı etkiler detaylı bir şekilde görselleştirilmiştir.

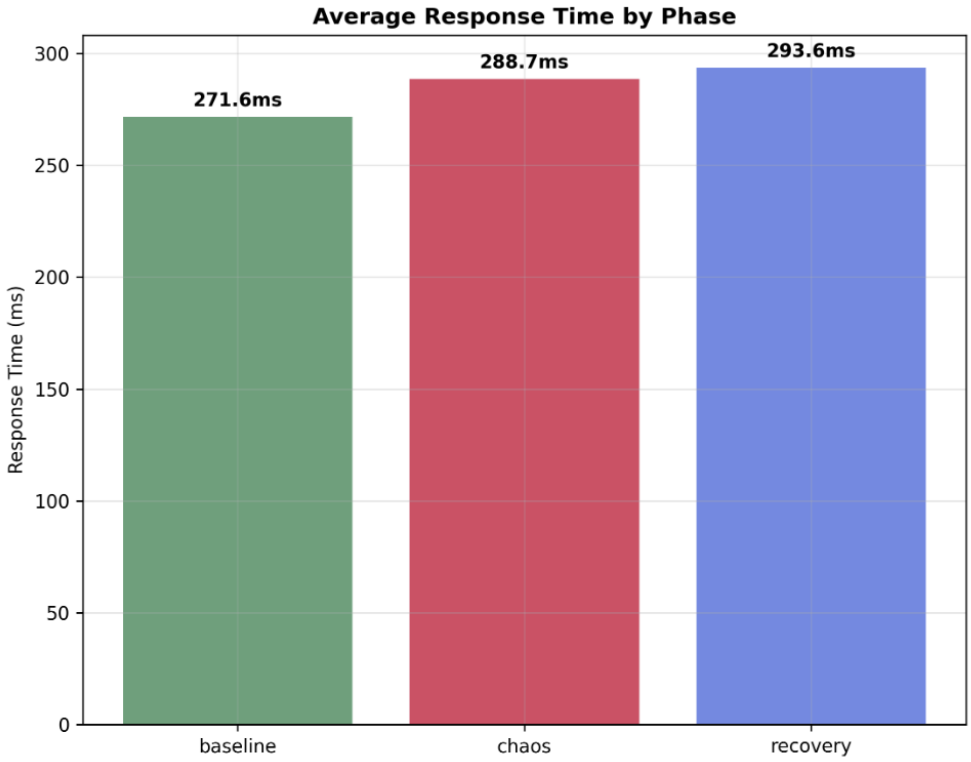


Şekil 4.11 Cpu Stres Servis Etki Analizi

Genel olarak elde edilen sonuçlar, CPU stresinin sistemde hata üretmekten çok, servislerin yanıt sürelerinde gecikmeye neden olduğunu; bu durumun ise özellikle yüksek yük anlarında kullanıcı deneyimini olumsuz etkileyebileceğini göstermektedir. Bu nedenle, mikroservis tabanlı yapılarda kaynak yönetimi ve performans takibi kritik bir öneme sahiptir.

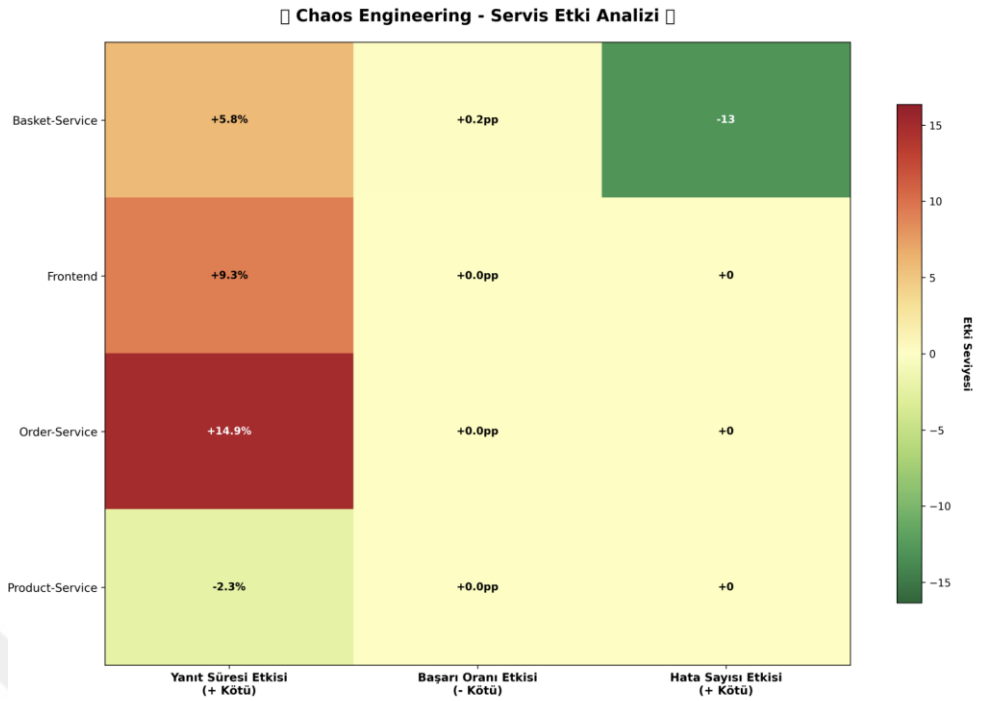
4.5 Bellek Stres Deney Sonuçları

Bellek stres testi süresince sistemin davranışı yakından izlenmiş ve özellikle yanıt sürelerinde Şekil 4.12 de gösterildiği üzere anlamlı değişiklikler gözlemlenmemiştir. Deneyin başlangıcında servislerin yanıt süreleri stabil seyrederken, bellek kullanımının kontrollü olarak artırılmasıyla birlikte, belirli aralıklarda yanıt sürelerinde çok belirgin olmayan yükselmeler yaşanmıştır.



Şekil 4.12 Bellek Stres Sistem Ortalama Cevap Süreleri

Deney boyunca sistemin hata toleransında önemli bir değişiklik meydana gelmemiş, hata oranı düşük seviyede kalmıştır. Buna rağmen, artan bellek kullanımı servislerin tepki hızını doğrudan etkilememiş ve yük altında sistemin yanıt verebilirliği bir miktar etkilenmiştir. Sistemin genelindeki etkiler, Şekil 4.13 te gösterilen kaos etki haritası ile analiz edilmiştir. Bu harita üzerinden, bellek stresinin yalnızca hedeflenen serviste değil, bağlı diğer servislerde de dolaylı etkiler oluşturduğu tespit edilmiştir. Özellikle yüksek bellek kullanımı anlarında, servisler arası etkileşimde gecikmeler ve performans farklılıkları daha belirgin hale gelmiştir.



Şekil 4.13 Bellek Stres Servis Etki Analizi

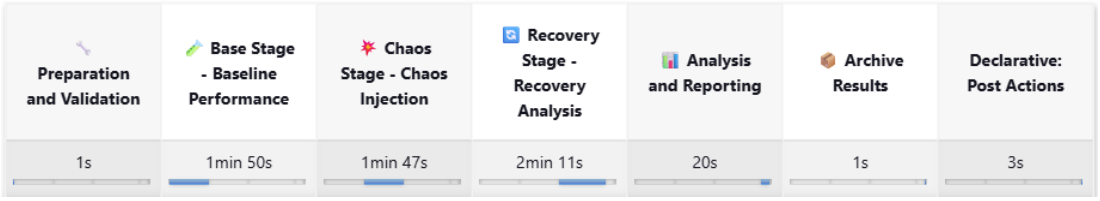
Elde edilen sonuçlar, yüksek bellek tüketimi altında sistemin kararlı kalabildiğini; ancak yanıt sürelerinde gecikmeler oluşabileceğini göstermektedir. Bu bulgular, kaynak yönetimi ve performans izleme süreçlerinin mikroservis tabanlı yapılarda ne kadar önemli olduğunu bir kez daha ortaya koymaktadır.

4.6 Deneylerin Yürütülmesi

Bu çalışmada, mikroservis mimarisi üzerinde gerçekleştirilen kaos deneylerinin tamamı otomatik olarak yürütülmüştür. Deneylerin otomasyonu için CI/CD süreçlerinde yaygın olarak kullanılan Jenkins aracı tercih edilmiş ve özel olarak hazırlanan bir pipeline tanımı üzerinden süreçler yönetilmiştir. Geliştirilen pipeline yapısı sayesinde, kaos deneyleri insan müdahalesi olmadan, tutarlı ve tekrarlanabilir bir şekilde uygulanabilmektedir.

Pipeline'ın başlangıcında, testin toplam süresi, hedef servis ve kaos deneyinin türü gibi parametreler kullanıcı tarafından belirlenmektedir. Ardından, çalışma alanı temizlenmekte ve gerekli tüm betikler, konfigürasyon dosyaları ve deney senaryoları ilgili dizinlere otomatik olarak kopyalanmaktadır. Deney süreci üç ana aşamadan oluşmaktadır: temel performans ölçümü (baseline)**,** kaos etkisi altında sistem davranışı

(chaos stage) ve toparlanma/recovery fazları. Her bir aşamada, JMeter ile trafik üretilmekte ve sistemin cevap süreleri, işlem hacmi ve başarı oranı gibi metrikler sürekli olarak toplanmaktadır. Kaos enjeksiyonu aşamasında, seçilen hedef servise yönelik önceden hazırlanmış YAML tabanlı kaos senaryosu, Şekil4.14 te gösterilen pipeline içerisinde dinamik olarak yapılandırılarak Kubernetes ortamına uygulanmaktadır.



Şekil 4.14 Pipeline Kurgusu

Böylece, ilgili mikroserviste belirli bir süre boyunca CPU stresi, bellek stresi, ağ gecikmesi veya pod terminasyonu gibi gerçekçi arızalar simüle edilmektedir. Aynı zamanda, deneyin her fazında JMeter ile yük testi gerçekleştirilmekte ve ortaya çıkan davranış verileri .jtl uzantılı dosyalarda toplanmaktadır.

Deneyin tamamlanmasının ardından, toplanan test sonuçları otomatik analiz betikleri ile işlenmektedir. Bu analizlerde, farklı deney fazlarının metrikleri karşılaştırılarak, sistemin arızalara karşı dayanıklılık seviyesi ve toparlanma süreci ortaya konulmaktadır. Ayrıca, her bir faz için elde edilen performans göstergeleri çeşitli grafikler ve özet tablolar halinde raporlanmaktadır. Tüm bu süreçlerin uçtan uca otomasyon ile yönetilmesi, hem zamandan tasarruf sağlamış hem de insan hatasını minimuma indirgemiştir. Bu bütünleşik otomasyon altyapısı sayesinde, mikroservis tabanlı sistemlerde kaos mühendisliği deneyleri yüksek doğruluk ve tekrarlanabilirlik ile yürütülmüş, elde edilen veriler ise hem sistemin zayıf noktalarını belirlemede hem de mimari kararların değerlendirilmesinde önemli bir rol oynamıştır.

5. TARTIŞMA

Bu çalışma kapsamında gerçekleştirilen kaos mühendisliği deneyleri, mikroservis mimarisine sahip sistemlerin dayanıklılığının sistematik ve otomatik test edilmesinin, hem yazılım kalitesi hem de operasyonel sürdürülebilirlik açısından kritik rol oynadığını göstermiştir [14]. Deneylerde, farklı hata senaryoları üretmek suretiyle sistemin kararlılığı ve beklenmeyen durumlara karşı tepkisi ölçülmüş, elde edilen bulgular mikroservislerin esnekliğini ve hata tolerans mekanizmalarını değerlendirme fırsatı sunmuştur [10]. Özellikle, pod terminasyonu, ağ gecikmesi ve stres testleri gibi çeşitli senaryolar üzerinden yapılan deneyler sayesinde, sistemin beklenen "steady-state" durumunun hangi koşullarda bozulduğu ve nasıl iyileştiği gözlemlenmiştir [18], [25]. Bu yaklaşım sayesinde, uygulamanın yalnızca teorik olarak değil, gerçek dünyadaki karmaşık ve beklenmedik koşullara da ne kadar dayanıklı olduğu anlaşılmıştır.

Yapılan analizlerde, sistemdeki hata oranlarının, tepki sürelerinin ve diğer servislere olan etkileri gibi performans metriklerinin değişimi incelenerek, mikroservis tabanlı uygulamalarda bulunan dayanıklılık paternlerinin pratikteki etkileri somut şekilde ortaya konmuştur. Deney sonuçları, sistemin hem kısa süreli hem de uzun süreli arızalara karşı esnekliğini anlamada değerli bilgiler sunmuştur. Ayrıca, otomasyonun getirdiği tekrarlanabilirlik ve tarafsızlık, insan hatasından kaynaklanabilecek sapmaların önüne geçilmesine olanak sağlamıştır. Sonuç olarak, mikroservis mimarisine sahip sistemlerde kaos mühendisliğinin düzenli olarak uygulanması, sistemin zayıf noktalarının tespit edilip güçlendirilmesine, kesintisiz hizmet sunumunun garanti altına alınmasına ve operasyonel risklerin azaltılmasına katkı sağlamaktadır.

5.1 Geçerliliğe Yönelik Şartlar ve Tehditler

Yürütülen kaos mühendisliği deneylerinin geçerliliği, deney ortamının gerçek üretim ortamını ne kadar yansıttığına, kullanılan araçların kapsamına, elde edilen verilerin nesnelliğine ve istatistiksel analizlerin doğruluğuna bağlıdır. Öncelikle, deneyler Minikube üzerinde kurulu bir mikroservis platformunda ve belirli kaynak kısıtları altında gerçekleştirilmiştir. Gerçek dünyadaki canlı sistemler ise çok daha büyük ölçeklerde çalışıyor olabileceğinden, ölçek ve çeşitlilik açısından farklılıklar,

dış geçerliliği sınırlayabilir. Ayrıca, kullanılan deney araçlarının (örneğin Chaos Mesh, Jenkins, JMeter) kendi sınırları ve ekosistemleri, elde edilen bulguların genellenebilirliğini etkileyebilir [20]. Deneylerin iç geçerliliği açısından, tekrarlanabilir senaryolar ve otomasyonun getirdiği standartlık önemli bir avantaj sunarken, sistem üzerinde yapılan yüklemelerin ve hata enjeksiyonlarının gerçek kullanıcı davranışlarını tam olarak yansıtıp yansıtmadığı sorgulanmalıdır [18]. Bazı hata türleri (örneğin donanım arızası, beklenmeyen altyapı problemleri) simüle edilse de, üretim ortamında ortaya çıkabilecek çok boyutlu ve nadir olaylar tam olarak modellenememiş olabilir. Ayrıca, deneyler sırasında kullanılan metriklerin seçimi, analiz edilen zaman aralıkları ve istatistiksel yöntemler de sonuçların güvenilirliğinde belirleyici faktörlerdir.

Bu tehditleri azaltmak için, literatürde önerilen farklı hata senaryoları ve yük profilleri dikkate alınmış, deneyler çoklu tekrarlarla desteklenmiş ve elde edilen veriler tarafsız biçimde analiz edilmiştir. Yine de, yapılan çalışmanın, belirli bir sistem konfigürasyonunda ve belirli teknolojiler kullanılarak elde edilen sonuçları içerdiği unutulmamalıdır. Farklı altyapıların, yazılım yığınlarının veya operasyonel pratiklerin kullanıldığı sistemlerde benzer deneylerin tekrarlanması, genellenebilirlik açısından önem taşımaktadır [11] [13].

6. SONUÇ

Bu çalışmada, mikroservis mimarisine sahip sistemlerde kaos deneylerinin otomatikleştirilmesi konusu ele alınmıştır. Gelişen yazılım dünyasında, sistemlerin karmaşıklığı ve dinamikliği, klasik test yöntemlerinin ötesine geçen yeni yaklaşımları gerektirmiştir. Bu kapsamda, kaos mühendisliği yaklaşımlarının, özellikle gerçek zamanlı canlı ortamlarda sistem esnekliğini ve dayanıklılığını artırmadaki rolü, gerçekleştirilen deneylerle kapsamlı biçimde incelenmiştir.

Deneylerde, pod terminasyonu, ağ gecikmesi, ağ kesintisi, CPU ve bellek stresi gibi tipik hata senaryoları, Minikube üzerinde çalışan bir e-ticaret uygulaması ve Chaos Mesh aracı ile otomatik şekilde tetiklenmiştir. Deneylerin planlanması, yürütülmesi ve sonuçların değerlendirilmesi süreçlerinde Jenkins pipeline'ları ve JMeter gibi otomasyon araçları etkin olarak kullanılmıştır. Her bir deney türü için sistemin normal ve hata anındaki davranışı karşılaştırılmış, elde edilen veriler istatistiksel olarak analiz edilmiştir. Yapılan analizler, deneysel süreçlerin otomatikleştirilmesinin, geleneksel manuel yöntemlere kıyasla daha güvenilir ve tekrarlanabilir sonuçlar sunduğunu göstermiştir.

Uygulanan otomatik kaos deneylerinin, sistemdeki gizli zayıflıkların tespit edilmesini kolaylaştırdığı ve müdahale süreçlerinin hızlandırılmasına katkı sunduğu görülmüştür. Elde edilen bulgular, mikroservis tabanlı sistemlerde dayanıklılığın artırılması için kaos mühendisliği uygulamalarının, modern yazılım geliştirme yaşam döngüsünün vazgeçilmez bir parçası olması gerektiğini ortaya koymaktadır. Ayrıca, deneylerin otomatikleştirilmesi sayesinde hem hata senaryoları çeşitlendirilebilmekte hem de olası riskler sistemli biçimde izlenebilmektedir.

Bu çalışma kapsamında, mikroservis mimarisi üzerinde toplamda dört farklı kaos deneyi pod terminasyonu, ağ gecikmesi, ağ kesintisi ve CPU ve bellek stres testi—otomatik olarak uygulanmıştır. Deneylerde, sistemin normal çalışma koşullarında 250 ms civarında olan ortalama yanıt süresi, CPU stres ve ağ gecikmesi deneylerinde 1000 ms'nin üzerine çıkmıştır. Özellikle CPU stres testi sırasında gecikmeler belirgin şekilde artarken, hata oranı ise düşük kalmıştır. Ağ kesintisi ve ağ gecikmesi deneylerinde ise, %10 ve üzeri paket kaybı veya yüksek gecikme anlarında, sistemdeki hata oranı ve servis yanıt süreleri ciddi şekilde artış göstermiştir. Pod terminasyonu

senaryosunda, Kubernetes'in otomatik iyileşme mekanizması sayesinde sistem kısa sürede toparlanmış, ancak bu süreçte 5xx hata oranlarında ve yanıt süresinde geçici bir yükselme yaşanmıştır. Tüm bu deneyler, mikroservis mimarisinin beklenmedik hata durumlarına karşı gösterdiği dayanıklılığı ve zayıf noktalarını sayısal olarak ortaya koymuş; otomatikleştirilen kaos mühendisliği süreçlerinin tekrarlanabilir ve karşılaştırılabilir sonuçlar üretebildiğini göstermiştir.

Sonuç olarak, yapılan bu çalışma ile mikroservis mimarisinde kaos deneylerinin otomatik olarak yürütülmesinin hem yazılım sistemlerinin güvenilirliğini artırdığı, hem de operasyonel süreçlerde esneklik sağladığı gösterilmiştir. Elde edilen veriler ışığında, kaos mühendisliği yaklaşımlarının, özellikle ölçeklenebilir ve yüksek erişilebilirlik gerektiren modern yazılım sistemlerinde daha yaygın şekilde benimsenmesi, hem akademik hem de endüstriyel uygulamalar için önemli bir gelecek vadetmektedir.

KAYNAKÇA

[1] C. T. Joseph ve K. Chandrasekaran, "Straddling the crevasse: A review of microservice software architecture foundations and recent advancements", *Softw. Pract. Exp.*, c. 49, sy 10, ss. 1448-1484, Eki. 2019, doi: 10.1002/spe.2729.

[2] F. Dai, H. Chen, Z. Qiang, Z. Liang, B. Huang, ve L. Wang, "Automatic Analysis of Complex Interactions in Microservice Systems", *Complexity*, c. 2020, ss. 1-12, Mar. 2020, doi: 10.1155/2020/2128793.

[3] A. Basiri *vd.*, "Chaos Engineering", *IEEE Softw.*, c. 33, sy 3, ss. 35-41, May. 2016, doi: 10.1109/MS.2016.60.

[4] K. A. Torkura, M. I. H. Sukmana, F. Cheng, ve C. Meinel, "CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure", *IEEE Access*, c. 8, ss. 123044-123060, 2020, doi: 10.1109/ACCESS.2020.3007338.

[5] P. Di Francesco, P. Lago, ve I. Malavolta, "Architecting with microservices: A systematic mapping study", *J. Syst. Softw.*, c. 150, ss. 77-97, Nis. 2019, doi: 10.1016/j.jss.2019.01.001.

[6] G. Blinowski, A. Ojdowska, ve A. Przybylek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation", *IEEE Access*, c. 10, ss. 20357-20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[7] N. Dragoni *vd.*, "Microservices: Yesterday, Today, and Tomorrow", içinde *Present and Ulterior Software Engineering*, M. Mazzara ve B. Meyer, Ed., Cham: Springer International Publishing, 2017, ss. 195-216. doi: 10.1007/978-3-319-67425-4_12.

[8] A. Megargel, V. Shankararaman, ve D. K. Walker, "Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example", içinde *Software Engineering in the Era of Cloud Computing*, M. Ramachandran ve Z. Mahmood, Ed., içinde Computer Communications and Networks. , Cham: Springer International Publishing, 2020, ss. 85-108. doi: 10.1007/978-3-030-33624-0_4.

[9] S. Newman, *Building Microservices*. O'Reilly Media, 2021. [Çevrimiçi]. Erişim adresi: https://books.google.com.tr/books?id=aPM5EAAAQBAJ

[10] D. Kesim, "Assessing Resilience of Software Systems by Application of Chaos Engineering – A Case Study", s. 187.

[11] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, ve V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices", içinde *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Haz. 2016, ss. 57-66. doi: 10.1109/ICDCS.2016.11.

[12] C. Rosenthal ve N. Jones, *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media, 2020. [Çevrimiçi]. Erişim adresi: https://books.google.es/books?id=iVjbDwAAQBAJ

[13] H. Chen, P. Chen, ve G. Yu, "A Framework of Virtual War Room and Matrix Sketch-Based Streaming Anomaly Detection for Microservice Systems", *IEEE Access*, c. 8, ss. 43413-43426, 2020, doi: 10.1109/ACCESS.2020.2977464.

[14] A. Basiri, L. Hochstein, N. Jones, ve H. Tucker, "Automating Chaos Experiments in Production", içinde *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May. 2019, ss. 31-40. doi: 10.1109/ICSE-SEIP.2019.00012.

[15] C. Camacho, P. C. Cañizares, L. Llana, ve A. Núñez, “Chaos as a Software Product Line—A platform for improving open hybrid-cloud systems resiliency”, *Softw. Pract. Exp.*, c. 52, sy 7, ss. 1581-1614, 2022, doi: 10.1002/spe.3076.
[16] O. Sharma, M. Verma, S. Bhadauria, ve P. Jayachandran, “A Guided Approach Towards Complex Chaos Selection, Prioritisation and Injection”, içinde *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, Tem. 2022, ss. 91-93. doi: 10.1109/CLOUD55607.2022.00025.
[17] S. De, “A Study on Chaos Engineering for improving Cloud Software Quality and Reliability”, içinde *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, Kas. 2021, ss. 289-294. doi: 10.1109/CENTCON52345.2021.9688292.
[18] C. Konstantinou, G. Stergiopoulos, M. Parvania, ve P. Esteves-Verissimo, “Chaos Engineering for Enhanced Resilience of Cyber-Physical Systems”, içinde *2021 Resilience Week (RWS)*, Eki. 2021, ss. 1-10. doi: 10.1109/RWS52686.2021.9611797.
[19] R. K. Lenka, S. Padhi, ve K. M. Nayak, “Fault Injection Techniques - A Brief Review”, içinde *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, Eki. 2018, ss. 832-837. doi: 10.1109/ICACCCN.2018.8748585.
[20] H. Jernberg, P. Runeson, ve E. Engström, “Getting Started with Chaos Engineering - Design of an Implementation Framework in Practice”, içinde *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, içinde ESEM ’20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3382494.3421464.
[21] A. Blohowiak, A. Basiri, L. Hochstein, ve C. Rosenthal, “A Platform for Automating Chaos Experiments”, içinde *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Eki. 2016, ss. 5-8. doi: 10.1109/ISSREW.2016.52.
[22] D. Kesim, A. van Hoorn, S. Frank, ve M. Häussler, “Identifying and Prioritizing Chaos Experiments by Using Established Risk Analysis Techniques”, içinde *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, Eki. 2020, ss. 229-240. doi: 10.1109/ISSRE5003.2020.00030.
[23] H. Chen, K. Wei, A. Li, T. Wang, ve W. Zhang, “Trace-based Intelligent Fault Diagnosis for Microservices with Deep Learning”, içinde *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, Tem. 2021, ss. 884-893. doi: 10.1109/COMPSAC51774.2021.00121.
[24] A. Nagarajan ve A. Vaddadi, “Automated Fault-Tolerance Testing”, içinde *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Nis. 2016, ss. 275-276. doi: 10.1109/ICSTW.2016.34.
[25] L. Zhang, B. Morin, P. Haller, B. Baudry, ve M. Monperrus, “A Chaos Engineering System for Live Analysis and Falsification of Exception-Handling in the JVM”, *IEEE Trans. Softw. Eng.*, c. 47, sy 11, ss. 2534-2548, Kas. 2021, doi: 10.1109/TSE.2019.2954871.
[26] L. B. Canonico, V. Vakeel, J. Dominic, P. Rodeghero, ve N. McNeese, “Human-AI Partnerships for Chaos Engineering”, içinde *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering*

Workshops, içinde ICSEW'20. New York, NY, USA: Association for Computing Machinery, 2020, ss. 499-503. doi: 10.1145/3387940.3391493.
[27] J. Zhang, R. Ferydouni, A. Montana, D. Bittman, ve P. Alvaro, "3MileBeach: A Tracer with Teeth", içinde *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA: Association for Computing Machinery, 2021, ss. 458-472. [Çevrimiçi]. Erişim adresi: https://ezproxy.iku.edu.tr:2444/10.1145/3472883.3486986
[28] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, ve R. Padhye, "Service-Level Fault Injection Testing", içinde *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA: Association for Computing Machinery, 2021, ss. 388-402. [Çevrimiçi]. Erişim adresi: https://ezproxy.iku.edu.tr:2444/10.1145/3472883.3487005
[29] A. van Hoorn, A. Aleti, T. F. Düllmann, ve T. Pitakrat, "ORCAS: Efficient Resilience Benchmarking of Microservice Architectures", içinde *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Eki. 2018, ss. 146-147. doi: 10.1109/ISSREW.2018.00-10.
[30] H. Tucker, L. Hochstein, N. Jones, A. Basiri, ve C. Rosenthal, "The Business Case for Chaos Engineering", *IEEE Cloud Comput.*, c. 5, sy 3, ss. 45-54, Haz. 2018, doi: 10.1109/MCC.2018.032591616.
[31] N. Brousse ve O. Mykhailov, "Use of Self-Healing Techniques to Improve the Reliability of a Dynamic and Geo-Distributed Ad Delivery Service", içinde *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Eki. 2018, ss. 1-5. doi: 10.1109/ISSREW.2018.00-40.
[32] K. A. Torkura, M. I. H. Sukmana, F. Cheng, ve C. Meinel, "Security Chaos Engineering for Cloud Services: Work In Progress", içinde *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*, Eyl. 2019, ss. 1-3. doi: 10.1109/NCA.2019.8935046.
[33] Z. Long, G. Wu, X. Chen, C. Cui, W. Chen, ve J. Wei, "Fitness-guided Resilience Testing of Microservice-based Applications", içinde *2020 IEEE International Conference on Web Services (ICWS)*, Eki. 2020, ss. 151-158. doi: 10.1109/ICWS49710.2020.00027.
[34] F. Poltronieri, M. Tortonesi, ve C. Stefanelli, "ChaosTwin: A Chaos Engineering and Digital Twin Approach for the Design of Resilient IT Services", içinde *2021 17th International Conference on Network and Service Management (CNSM)*, Eki. 2021, ss. 234-238. doi: 10.23919/CNSM52442.2021.9615519.
[35] N. Luo ve Y. Xiong, "Platform Software Reliability for Cloud Service Continuity - Challenges and Opportunities", içinde *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Ara. 2021, ss. 388-393. doi: 10.1109/QRS54544.2021.00050.
[36] N. Luo ve L. Zhang, "Chaos Driven Development for Software Robustness Enhancement", içinde *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, Ağu. 2022, ss. 1029-1034. doi: 10.1109/DSA56465.2022.00154.
[37] L. Zhang, B. Morin, B. Baudry, ve M. Monperrus, "Maximizing Error Injection Realism for Chaos Engineering with System Calls", *IEEE Trans. Dependable Secure Comput.*, ss. 1-1, 2021, doi: 10.1109/TDSC.2021.3069715.
[38] M. A. Naqvi, S. Malik, M. Astekin, ve L. Moonen, "On Evaluating Self-Adaptive and Self-Healing Systems using Chaos Engineering", içinde *2022*

IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), Eyl. 2022, ss. 1-10. doi: 10.1109/ACSOS55765.2022.00018.
[39] J. Simonsson, L. Zhang, B. Morin, B. Baudry, ve M. Monperrus, "Observability and chaos engineering on system calls for containerized applications in Docker", *Future Gener. Comput. Syst.*, c. 122, ss. 117-129, Eyl. 2021, doi: 10.1016/j.future.2021.04.001.
[40] A. Al-Said Ahmad ve P. Andras, "Scalability resilience framework using application-level fault injection for cloud-based software services", *J. Cloud Comput.*, c. 11, sy 1, s. 1, Oca. 2022, doi: 10.1186/s13677-021-00277-z.
[41] "The Observability, Chaos Engineering, and Remediation for Cloud-Native Reliability", içinde *Cloud-Native Computing*, John Wiley & Sons, Ltd, 2022, ss. 71-93. doi: 10.1002/9781119814795.ch4.

EK 1. SLR ÇALIŞMASINDA SEÇİLEN YAYINLAR

ID	Referans	Başlık	Yıl	Veritabanı
S1	[14]	Automating Chaos Experiments in Production	2019	ACM
S2	[20]	Getting Started with Chaos engineering - design of an implementation framework in practice	2020	ACM
S3	[26]	Human-AI Partnerships for Chaos engineering	2020	ACM
S4	[27]	3MileBeach: A Tracer with Teeth	2021	ACM
S5	[28]	Service-Level Fault Injection Testing	2021	ACM
S6	[21]	A Platform for Automating Chaos Experiments	2016	IEEE Xplore
S7	[24]	Automated Fault-Tolerance Testing	2016	IEEE Xplore
S8	[11]	Gremlin: Systematic Resilience Testing of Microservices	2016	IEEE Xplore
S9	[19]	Fault Injection Techniques - A Brief Review	2018	IEEE Xplore
S10	[29]	ORCAS: Efficient Resilience Benchmarking of Microservice Architectures	2018	IEEE Xplore
S11	[30]	The Business Case for Chaos engineering	2018	IEEE Xplore
S12	[31]	Use of Self-Healing Techniques to Improve the Reliability of a Dynamic and Geo-Distributed Ad Delivery Service	2018	IEEE Xplore
S13	[32]	Security Chaos engineering for Cloud Services: Work In Progress	2019	IEEE Xplore
S14	[13]	A Framework of Virtual War Room and Matrix Sketch-Based Streaming Anomaly Detection for Microservice Systems	2020	IEEE Xplore
S15	[4]	CloudStrike: Chaos engineering for Security and Resiliency in Cloud Infrastructure	2020	IEEE Xplore
S16	[22]	Identifying and Prioritizing Chaos Experiments by Using Established Risk Analysis Techniques	2020	IEEE Xplore
S17	[33]	Fitness-guided Resilience Testing of Microservice-based Applications	2020	IEEE Xplore
S18	[25]	A Chaos engineering System for Live Analysis and Falsification of Exception-Handling in the JVM	2021	IEEE Xplore
S19	[17]	A Study on Chaos engineering for Improving Cloud Software Quality and Reliability	2021	IEEE Xplore
S20	[18]	Chaos engineering for Enhanced Resilience of Cyber-Physical Systems	2021	IEEE Xplore
S21	[34]	ChaosTwin: A Chaos engineering and Digital Twin Approach for the Design of Resilient IT Services	2021	IEEE Xplore
S22	[35]	Platform Software Reliability for Cloud Service Continuity - Challenges and Opportunities	2021	IEEE Xplore

S23	[23]	Trace-based Intelligent Fault Diagnosis for Microservices with Deep Learning	2021	IEEE Xplore
S24	[16]	A Guided Approach Towards Complex Chaos Selection, Prioritisation and Injection	2022	IEEE Xplore
S25	[36]	Chaos Driven Development for Software Robustness Enhancement	2022	IEEE Xplore
S26	[37]	Maximizing Error Injection Realism for Chaos engineering With System Calls	2022	IEEE Xplore
S27	[38]	On Evaluating Self-Adaptive and Self-Healing Systems using Chaos engineering	2022	IEEE Xplore
S28	[39]	Observability and chaos engineering on system calls for containerized applications in Docker	2021	ScienceDirect
S29	[40]	Scalability resilience framework using application-level fault injection for cloud-based software services	2022	Springer
S30	[15]	Chaos as a Software Product Line—A platform for improving open hybridcloud systems resiliency	2022	Wiley
S31	[41]	The Observability, Chaos engineering, and Remediation for Cloud-Native Reliability	2022	Wiley