T.C.
İSTANBUL ÜNİVERSİTESİ-CERRAHPAŞA
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

FARKLI PLATFORMLARDAKİ ÖLÇEKLENEBİLİR WEB UYGULAMA PROGRAMLAMA ARAYÜZLERİNİN PERFORMANS AÇISINDAN KARŞILAŞTIRILMASI

Erdem KEMER

DANIŞMAN
Doç. Dr. Rüya ŞAMLI

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

İSTANBUL-2019

Bu çalışma 26.03.2019 Tarihinde aşağıdaki jüri tarafından Bilgisayar Mühendisliği Anabilim Dalı, Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı Yüksek Lisans Tezi olarak kabul edilmiştir.

TEZ JÜRİSİ

Doç. Dr. Rüya ŞAMLI
İstanbul Üniversitesi-Cerrahpaşa
Mühendislik Fakültesi

Prof. Dr. Ahmet SERTBAŞ
İstanbul Üniversitesi-Cerrahpaşa
Mühendislik Fakültesi

Dr.Öğr.Üyesi Zeynep TURGUT
Haliç Üniversitesi
Mühendislik Fakültesi

ÖNSÖZ

Web servislerinin çeşitlerini, özelliklerini ve performans karşılaştırmalarını araştırıp raporlamış olduğum bu tezi dikkatinize sunmaktan onur duyarım.

Çalışmanın hazırlanması sürecinde hiçbir desteği esirgemeyen değerli hocam Doç. Dr. Rüya ŞAMLI'ya, sabırları ve sevgileri için değerli eşim Fazilet KEMER başta olmak üzere, aileme ve arkadaşlarıma teşekkür ederim.

Mart 2019 Erdem KEMER

İÇİNDEKİLER

Sayfa No

ŞEKİL LİSTESİ

Sayfa No

TABLO LİSTESİ

Sayfa No

SİMGE VE KISALTMA LİSTESİ

Kısaltmalar	Açıklama
API	: Application Programming Interface – Uygulama Programlama Arayüzü
BPML	: Business Process Modeling Language – İş Süreçleri Modelleme Dili
BPMN	: Business Process Modeling Notation – İş Süreçleri Modelleme Notasyonu
BT	: Bilgi Teknolojileri
CLI	: Common Language Infrastructure – Ortak Dil Altyapısı
CLR	: Common Language Runtime – Ortak Çalışma Zamanı
DNS	: Domain Name System – Alan Adı Sistemi
GIS	: Geographic Information System – Coğrafî Bilgi Sistemi
GRASS	: Geographic Resources Analysis Support System – Coğrafi Kaynaklar Analiz Destek Sistemi
GUI	: Graphical User Interface – Grafiksel Kullanıcı Arayüzü
HATEOAS	: Hypermedia as the Engine of Application State – Uygulama Durumunun Motoru Olarak Hipermedya
HMM	: Hidden Markov Model – Gizli Markov Modeli
HTTP	: HyperText Transfer Protocol – HiperMetin Aktarma Protokolü
IIS	: Internet Informaton Service – Internet Bilgi Hizmetleri
IL	: Intermediate Language – Ara Dil
IP	: Internet Protocol – İnternet Protokolü
IOT	: Internet of Things – Nesnelerin İnterneti
JSON	: JavaScript Object Notation – JavaScript Nesne Notasyonu
JVM	: Java Virtual Machine – Java Sanal Makinesi
LB	: Load Balancer – Yük Dengeleyici
MSIL	: Microsoft Intermediate Language – Microsoft Ara Dili
P2P	: Peer to Peer – Eşler Arası
QoS	: Quality of Service – Hizmet Kalitesi
REST	: Representational State Transfer – Temsili Durum Transferi
RPC	: Remote Procedure Call – Uzaktan Yordam Çağrısı
SaaS	: Software as a Service – Servis Olarak Yazılım

SBML : Systems Biology Markup Language – Sistem Biyolojisi Biçimlendirme Dili

SAN : Storage Area Network – Depolama Alan Ağı

SOA : Service Oriented Architecture – Servis Odaklı Mimari

SOAP : Simple Object Access Protocol – Basit Nesne Erişim Protokolü

SSL : Secure Sockets Layer – Güvenli Soket Katmanı

SUT : Software Under Test – Test Altındaki Yazılım

TCP : Transmission Control Protocol – İletişim Kontrol Protokolü

TLS : Transport Layer Security – Taşıma Katmanı Güvenliği

UDDI : Universal Description, Discovery and Integration – Evrensel Açıklama, Keşif ve Tümleştirme

UDP : User Datagram Protocol – Kullanıcı Datagram Protokolü

UML : Unified Modeling Language – Tümleşik Modelleme Dili

URI : Uniform Resource Identifier – Tekdüzen Kaynak Tanımlayıcısı

URL : Uniform Resource Locator – Tekdüzen Kaynak Konum Belirleyicisi

WS : Web Service – Web Servisi

WSDL : Web Services Description Language – Web Servisi Açıklama Dili

WWW : World Wide Web – İnternet Sunucuları Ağı

XML : Extensible Markup Language – Genişletilebilir Biçimlendirme Dili

ÖZET

YÜKSEK LİSANS TEZİ

FARKLI PLATFORMLARDAKİ ÖLÇEKLENEBİLİR WEB UYGULAMA PROGRAMLAMA ARAYÜZLERİNİN PERFORMANS AÇISINDAN KARŞILAŞTIRILMASI

Erdem KEMER

İstanbul Üniversitesi-Cerrahpaşa

Lisansüstü Eğitim Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalı

Danışman : Doç. Dr. Rüya ŞAMLI

İnternete bağlanabilen cihazların sayısı gün gittikçe artmaktadır. İnsanlar tarafından sabit bilgisayarlardan bilgi edinme ihtiyacını karşılamak için başlayan internet kullanımı, sonrasında taşınabilir bilgisayarlar ve mobil telefonlar ile daha ulaşılabilir bir hale gelmiştir. Günümüzde akıllı ev yönetim sistemleri ve birçok cihaz, internet üzerinden bağlantı kurar hale gelmiştir. Artık günlük hayatın içine kadar giren bu cihazlar sayesinde birçok iş çevrimiçi olarak internet üzerinden, bilgisayar ve diğer cihazları kullanılarak yapılabilmektedir. Özellikle, Nesnelerin Interneti (Internet of Things – IOT) kavramından sonra web uygulama programlama arayüzleri (WebAPI) çok daha büyük bir öneme kavuşmuştur. Günümüzde, cihazların, sistemlerin ve modüllerin birbirleri ile iletişimleri büyük oranda WebAPI'ler üzerinden yapılmaktadır. Bu bağlamda WebAPI'lerin performans ve ölçeklenebilirliği çok büyük önem kazanmıştır. Bu tez kapsamında farklı programlama dilleri ile aynı işi yapan örnek prototip WebAPI'ler geliştirilecek olup, bunların performansları birbiri ile kıyaslanacaktır.

Mart 2019, 95 sayfa.

Anahtar kelimeler: Web Servisleri, Nesnelerin İnterneti, Performans Analizi

SUMMARY

M.Sc. THESIS

PERFORMANCE COMPARISON OF SCALABLE WEB APPLICATION DEVELOPMENT INTERFACES IN DIFFERENT PLATFORMS

Erdem KEMER

Istanbul University-Cerrahpasa

Institute of Graduate Studies

Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Rüya ŞAMLI

The number of devices that connect the internet is increasing each day. People started to connect the internet using their personal computers with the purpose of gathering information, then internet became more accessible with the increased popularity of portable computers and mobile phones. Today, intelligent home management systems and many other devices have become interconnected over the internet. As the result of these devices being used in every aspect of our lives, we can accomplish many of our daily tasks, easier than ever. We call these, internet connected devices, Internet of Things (IOT). As IOT become more and more popular, web application programming interfaces (WebAPIs) are now more important than ever. Today all these devices, systems and modules are mostly integrated over WebAPIs. Thus, performance and scalability features of the WebAPIs are very important. In the scope of this thesis, sample prototype WebAPIs will be developed using different programming languages and the performances of these WebAPIs will be benchmarked against each other.

March 2019, 95 pages.

Keywords: WebAPI, Internet of Things, Performance Analysis

1. GİRİŞ

Günümüzde şirketlerin karşı koyması gereken zorluklar, teknik altyapılarını etkilemiş ve heterojen yapıda dağılmış ortamlarda daha gevşek bağlanmış bileşenlere ihtiyaç duyulmaya başlanmıştır. Servis Odaklı Mimari (Service Oriented Architecture – SOA), bu gevşek bağı kolaylaştıran ve aynı zamanda kabul edilebilir çözümler için gerekli olan yeterli servis kalitesini sağlayan bir yaklaşımı temsil etmektedir [1]. Bu yaklaşım, kurumsal Bilgi Teknolojileri (BT) altyapıları için bir temel yapı taşıdır ve farklı işletmelerin gereksinimlerini ve ihtiyaçlarını yöneterek iş süreçlerini gerçekleştirmek için en çok kullanılan yaklaşımdır. Yapılan bazı araştırmalara göre, işletmelerin büyük bir kısmı SOA konusunda mevcut çözümlere sahipken, bir bölümü de konuyu yakın gelecekte ele almayı planlamaktadır. Şirketler için önemli bir konu olan SOA'nın temel bileşeni, "kendi kendine yeten ve kendi kendini tanımlayabilen gevşek bağlanmış, tekrar kullanılabilir bir yazılım bileşeni olan bir web servisi'dir [2].

Web servisleri, günümüzde servis odaklı olarak gelişen bilgi işlem dünyasının gereksinimlerini yerine getirmenin en önemli imkanlarından biridir. Web servisi, en genel tanımıyla "standart web protokolleri kullanılarak sistemlere yerleştirilebilen, yayınlanabilen ve çağrılabilen bir arayüze sahip bir bilgisayar programıdır" [3]. Başka bir tanımda ise web servisi, "bir ağ üzerinde birlikte çalışabilen makine – makine etkileşimini desteklemek için tasarlanmış bir yazılım sistemi" olarak geçmektedir. Bu tez çalışmasında, farklı platformlardaki ölçeklenebilir web servislerinin çeşitleri, özellikleri, uygulama alanları ve performans karşılaştırılması anlatılmaktadır.

Tez çalışmasının organizasyonu şu şekildedir: 2. bölümde tez çalışmasının konusu açıklanmış ve konu ile ilgili literatür taraması gerçekleştirilmiştir. 3. bölümde tez çalışmasının gerçekleştirilmesi için izlenen yöntemler, kullanılan veri ve araçlar tanıtılmıştır. 4. bölümde, giriş bölümünde açıklanan ön bilgilere göre elde edilmiş bulgular açıklanmıştır. 5. bölümde ise araştırma sonucunda ortaya çıkan bulgular ve elde edilen sonuçlar irdelenmiş ve ortaya çıkan fayda ve kazanımlar ifade edilmiştir.

2. GENEL KISIMLAR

Bu bölümde web servislerinin tanımı, özellikleri, kullanım yöntemleri ve çeşitleri anlatılmaktadır.

2.1. WEB SERVİSLERİ

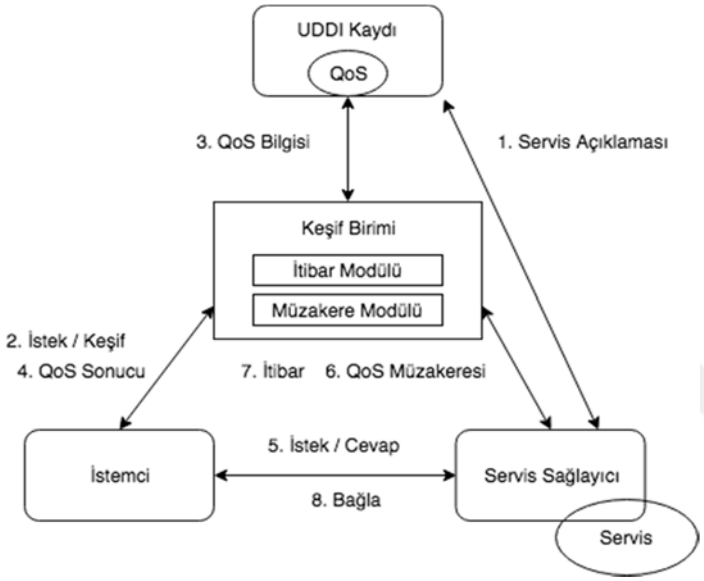
Günümüzün bilişim dünyasında, yazılımlar artık internet üzerindeki servisler olarak geliştirilmektedir. "İsteğe Bağlı Yazılım" veya Servis Olarak Yazılım (Software as a Service – SaaS) terimleriyle adlandırılan ilişkili dağıtım modeli, uygulamaların artık yükleme veya el ile yükseltme gerektirmediğini gösterir.

Internet sunucuları ağı (World Wide Web – WWW), geliştirildiği günden itibaren, isteğe bağlı yazılımların geliştirilmesini sağlamak için bir dizi aşamadan geçmiştir. Web sayfaları, başlangıçta, sınırlı kullanıcı etkileşimi kabiliyetlerine sahip basit metin belgelerinden biraz daha fazlasıydı; yalnızca grafik desteği ve form tabanlı veri girişi yapılabilen bağlantılar ve tam sayfa güncellemelerine dayanan uygulamalara izin vermekteydi. Ardından, zaman içerisinde, sayısız interaktif internet teknolojisi ile, gelişmiş web ve grafikler için yerleşik destekle giderek artan etkileşimli web sayfaları oluşturmak mümkün hale gelmiştir. Günümüzde, Web 2.0 teknolojileriyle, ağın yetenekleri ve yaygınlığı üzerine inşa ederek daha karmaşık uygulamalara doğru adımlar atılmaktadır. Sistemleri isteğe bağlı olarak indirme olanağı, geleneksel modellere göre pek çok yarar sağlamaktadır. İsteğe bağlı uygulamalar kullanıcılar tarafından kurulmaya ihtiyaç duymaz ve bu sayede, daha esnek bir şekilde yükseltilebilir. Ayrıca, aynı servis birçok kullanıcıya yaygın olarak sunulduğundan, oluşabilecek komplikasyonların, geleneksel modele göre çözülmesi çok daha kolaydır. Bunun yanısıra, servis sağlayıcı tüm verilere ek olarak kullanıcıların davranışlarına da erişebilir; bu da sistemin en önemli yönlerine odaklanan kapsamlı test sistemleri tasarlamayı kolaylaştırır. İsteğe bağlı uygulamaların diğer bir özelliği, web üzerinden genellikle internete yüklenmeleri ve web tarayıcısının çalışma ortamı olarak işlev görmesidir. Sonuç olarak, uygulamalar genellikle diğer işlemler için de kullanabilecekleri HTTP (HyperText Transfer Protocol – Hipermetin Aktarma Protokolü), REST (Representational State Transfer – Temsili Durum Transferi), SOAP (Simple Object Access Protocol – Basit Nesne Erişim Protokolü) ve JSON (JavaScript Object Notation – JavaScript Nesne Notasyonu) gibi web teknolojilerine dayanan uygulama programlama arayüzleri (Application Programming Interface – API) üzerine inşa edilir. Bu özellik, halihazırda var olan

içeriği ve servisleri tekrar tekrar kullanan, giderek karmaşıklaşan üçüncü parti uygulamaların geliştirilmesini sağlar. Çeşitli uygulamalardan gelen içeriği entegre bir deneyime dönüştüren ve "karma uygulama" olarak adlandırılan bu uygulamalar, yeniden kullanılan servislerin orijinal geliştiricileri ile doğrudan ilişkili olmayan geliştiriciler tarafından oluşturulabilir [4].

2.1.1. Web Servislerinin Keşfi

Geleneksel web servisleri yayın ve keşif modelinin üç rolü vardır: servis sağlayıcı, servis istemcisi ve Evrensel Tanım, Keşif ve Entegrasyon (Universal Description, Discovery and Integration – UDDI) kaydı. Şekil 2.1.'de gösterildiği gibi, model şu şekildedir:



Şekil 2.1: İtibara bağlı web servisi keşif ve seçim modeli

Şekildeki işlemlerin açıklaması şu şekildedir:

(1) Servis sağlayıcı, UDDI kayıtlarındaki servis açıklamasını yayınlar.

(2) İstemci, keşif birimindeki kalite matrisleriyle eşleşen bir servis ister.

(3) Keşif birimi, talebi istemci tarafından alır ve eşleşen bir servis arar ve ardından kendi gereksinimlerine uyan servisleri geri döner.

(4) İstemci, keşif biriminden bir web servisleri listesi olarak sonuç alır.

(5) İstemci, keşif biriminden arama sonucunu aldıktan sonra servis sağlayıcıdan hizmet ister.

(6) Müzakere modülü, istemci gereksinimlerini karşılayan bir web servisi bulamaması durumunda, müzakereyi kabul eden web servislerinin bir listesi hakkında müzakere yapabilir.

İstemci hizmeti aldıktan sonra, (7) servisler için bir itibar puanı atayabilir. Daha sonra itibar modülü, servisin yeni itibar değerini istemci tarafından atanan değere, istemcinin güven faktörüne ve servisin önceki itibar değerine göre hesaplar.

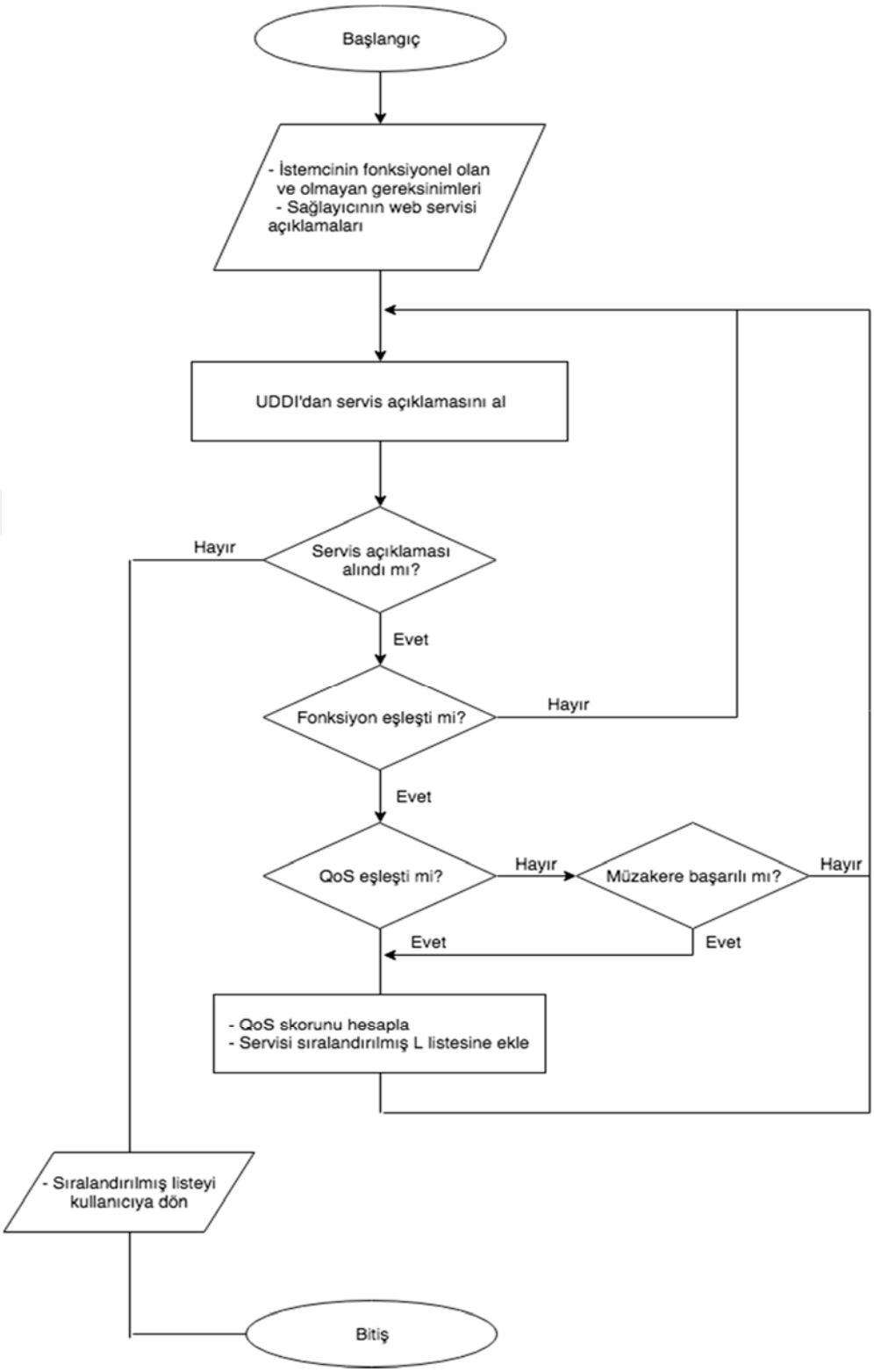
(8) Müşteri hizmeti alabilir [5].

Şekil 2.2, keşif yaklaşımının bir akış şemasını göstermektedir. Bu şemada, giriş, istemcinin işlevsel ve gereksinimlerini ifade eder. Keşif birimi keşif istekleri alır; UDDI kayıt defterinde yer alan işlevsel gereklilikleri kontrol eder ve işlevsel gereksinimleri karşılayan servislerin listesini verir. Müzakere modülü, müzakereyi kabul eden servislerle müzakere yapabilir ve eski listeye eklenmiş servislerin listesini döndürür. Keşif birimi web servislerinin istemci gereksinimlerini karşılayamaması durumunda, müzakere modülü, müzakere şartlarını kabul eden bir WS (Web Service – Web Servisi) listesi hakkında görüşme yapabilir. Müzakere başarılı olduktan sonra keşif birimi istemci gereksinimlerini karşılayan bir servis listesi verir; eğer müzakere başarılı olmazsa, hiçbir servis geri dönmez.

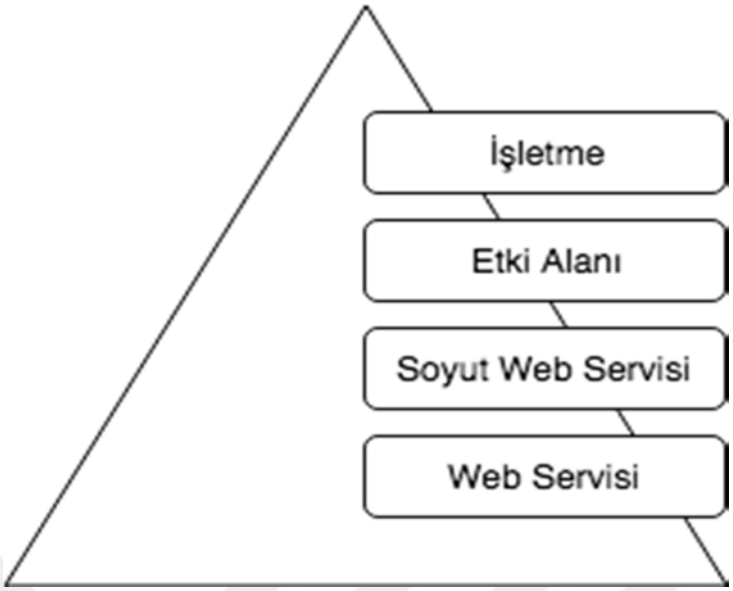
UDDI, web servisi ortamında çok önemli bir rol oynar. Servisleri yayımlamak ve bulmak için kullanılabilir. Mevcut UDDI modellerinin çoğu merkezidir. Bu nedenle, listelenen veya görüşme yapılan çok fazla hizmet varsa performansları düşer. Merkezi yapı daha az sağlamdır ve yetersiz birlikte çalışabilirliği ifade eder. Mevcut UDDI bir pasif dizin servisidir; bu, servis değişimlerini algılamayı zorlaştırır. Aşağıdaki sorular, problemi ifade etmektedir:

- İhtiyaçların tanımlanması: bulmak istediğimiz nedir, iyi ifade edilmiş mi? İyi yorumlanmış mı?
- Bilgilerin tanımlanması: mevcut olan nedir; iyi sunulmuş mu? Bulunabilmesi için iyi sınıflandırılmış mı?

Geleneksel bilgi alma sorunu bir sınıflandırma sorunu olarak çözülür. Servisler ne kadar iyi bir biçimde sınıflandırılırsa, onları birbirinden ayırmak ve keşfetmek daha kolay olacaktır. Sınıflandırma, yapay zeka problemlerini çözmek için benimsenen çözümlerden biridir. Keşif problemi, sınıflandırmanın bir çözüm olarak kullanılabileceği bir yapay zeka problemi olarak görülmektedir. Web servislerinin gösterimi için 4 seviyeli bir piramit modeli önerilmektedir (Şekil 2.3).



Şekil 2.2: İtibara bağlı web servisi keşfi akış şeması

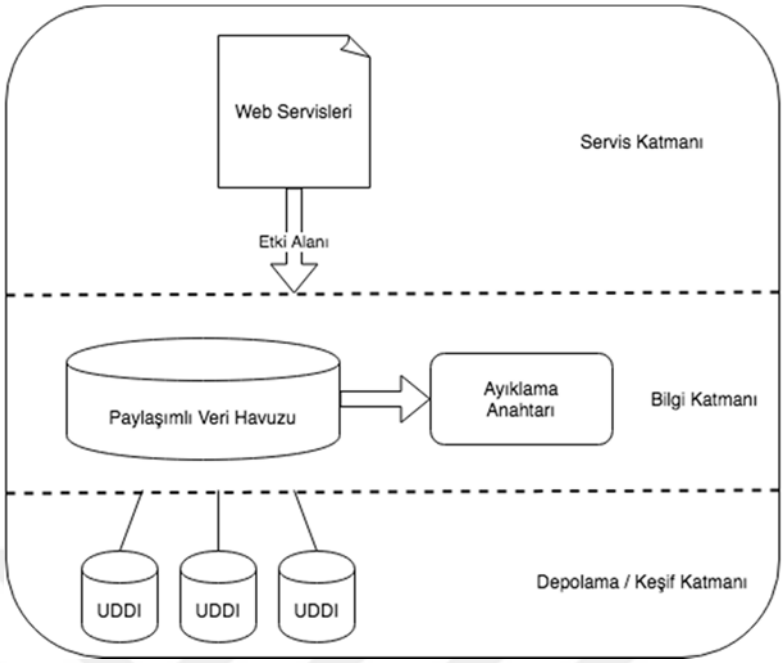


Şekil 2.3: Web servisi gösterimi için piramit modeli

Buradaki seviyeler şu şekilde açıklanabilir:

- İşletme seviyesi: Her bir kuruluşun birkaç etki alanı ifade edebileceği seviye.
- Etki alanı seviyesi: Her alanın bir anahtar ile tanımlandığı ve çeşitli özet servisleri içeren seviye.
- Soyut web servisi seviyesi: Her bir özet web servisinin bir anahtar ile tanımlandığı ve çeşitli yayınlanan web servislerini içeren seviye.
- Web servis seviyesi: Her web servisinin çift tarafından tanımlandığı seviye.

Katmanlar, etki alanı ve özet web servisi, web servislerinde bilgi gösterimi için paylaşılan depoyu (servislerin ontolojisini) temsil eder. Aynı depo, yayın, depolama ve keşif için kullanılacaktır. Web servisleri yayınlama (depolama) ve keşif sistemimizin işlevsel mimarisi üç katmana dayanmaktadır (Şekil 2.4) [6]. Bu katmanlar, servis katmanı, bilgi katmanı ve depolama/keşif katmanı şeklindedir. Bu katmanlı gösterim, sistemin kolayca tasarlanabilmesine, gelişimin karmaşıklığının azaltılmasına ve her bir katmanın bağımsız gelişimini sağlamaya olanak verecektir.



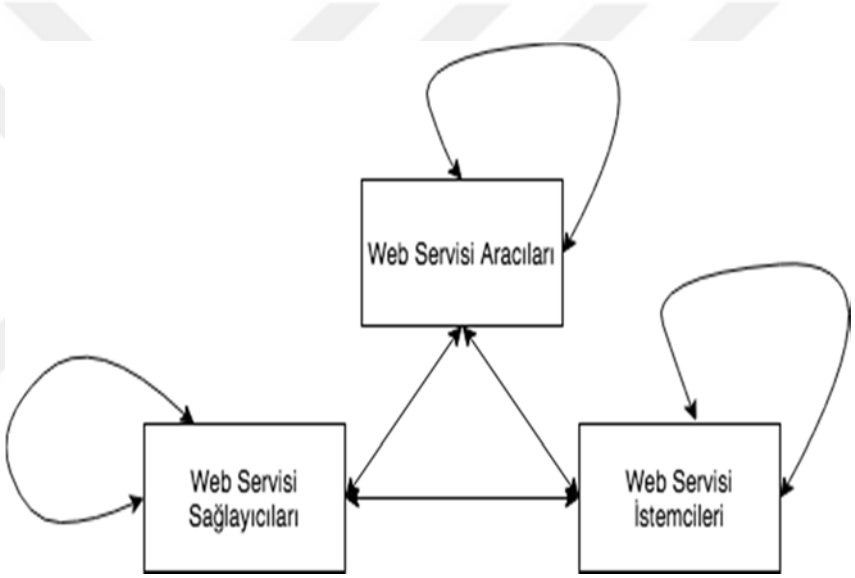
Şekil 2.4: Önerilen sistemin işlevsel mimarisi

2.1.2. Web Servislerinin Güvenliği

SOA, pek çok servis arasında geniş kapsamlı bir birlikte çalışabilirlik gerektirir. Bu da pek çok güvenlik sorununun ortaya çıkması ile sonuçlanan bir durumdur. Web servisinin mantıksal işlevselliğinin geliştirilmesi esnasında, güvenlik özellikleri planlanırsa, web servisleri son derece karmaşık hale gelir ve performansı ve ölçeklenebilirlik özellikleri büyük ölçüde azalır. Bu açıdan web servislerinin güvenlik sorunları, servis işlevlerinin dışında olmalıdır. Bu şekilde, hem tasarımın basitleşir ve web servisleri rahat bir şekilde çağrılır, hem de mesajlaşmanın güvenliğini sağlanmış olur.

Bir uygulama sistemi genellikle bir servis platformuna dayanır ve bu platformun genellikle kendi güvenlik çözümleri bulunmaktadır. Bir mesajın imzalanması için sertifika kullanma işlemi, servis istemcisi için aynı platformda imza doğrulaması yapmak ve servis sağlayıcısını imzalamak suretiyle gerçekleştirilebilir. Buna göre, servis istemcisi ve sağlayıcı farklı platformlarda ise, birlikte çalışabilirlik güvenliği garanti edilemez. Her uygulama platformunun kendine özgü bazı güvenlik mekanizmaları ve güvenlik API'si bulunmaktadır. Bu sayede farklı şekillerde güvenlik sağlanabilir. Web servislerinin heterojen platformdaki güvenlik mekanizmaları, mesajların güvenlik ilkesi yapılandırmasını sağlar. Güvenlik servisi araçları, mesajların güvenlik gereksinimlerinin "mesaj bütünlüğü, mesaj gizliliği ve mesaj doğruluğu" yönlerini elde etmeye çalışmaktadır [7].

Bir cihaza ya da servise olan güven, kanıtlara, deneyime ve algıya dayalı bir inanç olarak ifade edilmektedir. Ayrıca, güven, bir tarafın belirli bir ortakla belirli bir eylemde bulunmaya istekli olduğu risk ve teşviki dikkate alması olarak da tanımlanır. Web servislerinde, web servis sağlayıcısı veya web servis sorumlusu, güvenen veya güvenilen taraf olabilir. Web servislerinde herhangi iki aracı arasında güçlü veya zayıf bir güven ilişkisi bulunmaktadır. Şekil 2.5'te, web servislerinde çok ortaklı bir güven ilişkisi modeli gösterilmektedir. Tüm elemanlar arasında güven ilişkileri bulunabilir, ancak bu güven ilişkisinin güven değeri düşük veya yüksek olabilir [8]. Bu şekilde gösterilen güven talepleri, web servisleri ve e-ticaret alanlarında büyük ilgi görmüştür. Web servis sağlayıcı güveni, web servislerinde e-işlemin muadilindeki güven algısını ifade eder.



Şekil 2.5: Web servislerinin güven ilişkisi modeli

Web servis istemcisinin web servis sağlayıcılarına olan güveni, istemci odaklı web servisleri benimseme kararlarında önemlidir. Bu şekilde, döngüsel oklar olarak gösterilen bir güven ilişkisi vardır. Bu güven ilişkilerinin her biri, web servislerinin gelişimini etkiler. Örneğin, bir web servis sorumlusu, satın aldığı akıllı telefonu kullanıp bunu bir başkasına önerirse ve önerdiği kişi bu web servis sorumlusuna güvenirse, önerdiği kişi, web servisleri aracılığıyla telefon satın almak için başka bir web servis talebinde bulunur.

Web servis istemcileri, web servis sağlayıcıları ve web servis komisyoncuları, bir e-işlemi tamamlamak için genellikle birlikte çalışırlar. Ancak, bu durum güven sorunlarıyla karşı karşıya olmalarına sebep olabilmektedir. Bu elemanlar, bazen birbirlerine tamamen

güvenirken; bazen hiçbirine güvenmezler; bazen de kısmî bir güven söz konusu olur. Elemanlar arasında güven sağlanmışsa, web servisini tatmin edici bir şekilde tamamlamaları kolaylaşır, güven sağlanmamışsa, elemanlardan birisi web servisinden haksız bir menfaat elde edebilir. Web servislerindeki aracılar arasındaki güven ilişkileri, "güven ağı" olarak adlandırılabilecek bir ağ oluşturur. Şekil 2.5'teki güven ağlarına bakılacak olursa,

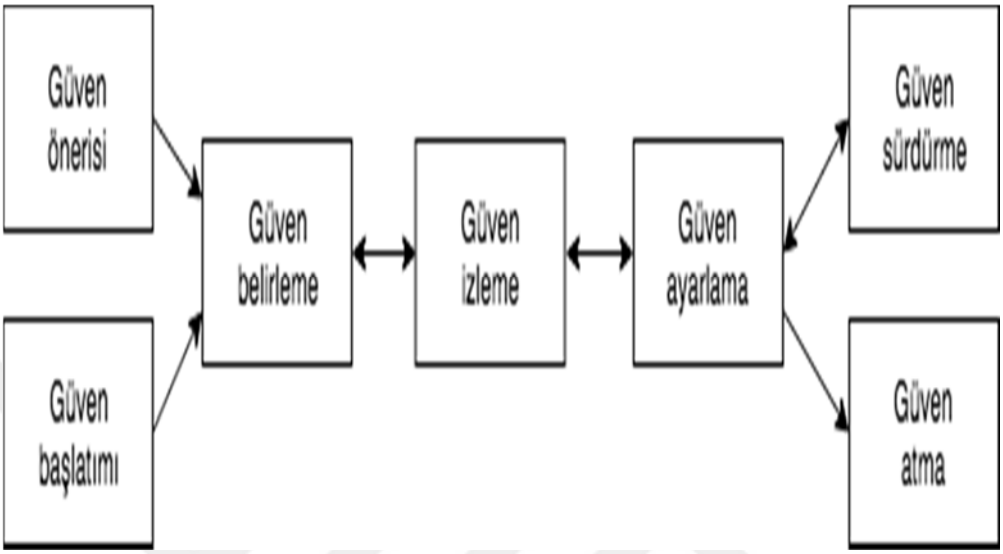
- web servis komisyoncuları arasında,
- web servis sağlayıcıları arasında,
- web servis istemcileri arasında,
- web servis komisyoncuları ve web servis istemcileri arasında,
- web servis istemcileri ve web servis sağlayıcıları arasında,
- web servis sağlayıcıları ve web servis komisyoncuları arasında

olmak üzere altı adet güven ağı bulunduğu görülebilir.

"Güven yönetimi", e-ticaret işlemleriyle ilgili güven değerlendirmelerini ve güven kararlarını vermek için güvenlikle ilgili kanıtların toplanması, kodlanması ve analizi olarak tanımlanmaktadır. Güven yönetimi, mevcut bir güven ilişkisini izlemek ve/veya yeni bir güven ilişkisi kurmak için gerekli olan bilgilerin toplanmasını gerektirir. Web servislerinde güven yönetiminin hiyerarşik bir yapısı Şekil 2.6'da gösterilmektedir. Bu yapı, güven yönetimi anlayışını geliştirmiştir.

Şekil 2.6: Web servislerinde güven yönetiminin hiyerarşik yapısı

Web servisleri için bir güven yönetim sistemi, bir web servisi istemcisi ve sağlayıcısının birbirleriyle tanışmalarını ve böylece kayıpları en aza indirmelerini sağlar. Şekil 2.7'de web servislerinde güven yönetiminin stratejik bir modeli verilmiştir.



Şekil 2.7: Web servislerinde güven yönetiminin stratejik bir modeli

Bu modeldeki bazı kavramlar aşağıda açıklanmıştır:

Güven izleme: Güven geçersiz kalana kadar devam eden bir eylemdir. Çünkü bir aracı her zaman web servislerinde azami yarar ya da menfaat sağlamaya çalışır. Güvenlik sağlandıktan bir süre sonra, genellikle güvensizlik oluşur. Bu nedenle, güven izleme, web servislerinde güven ilişkisini sürdürmek için stratejik bir görevdir.

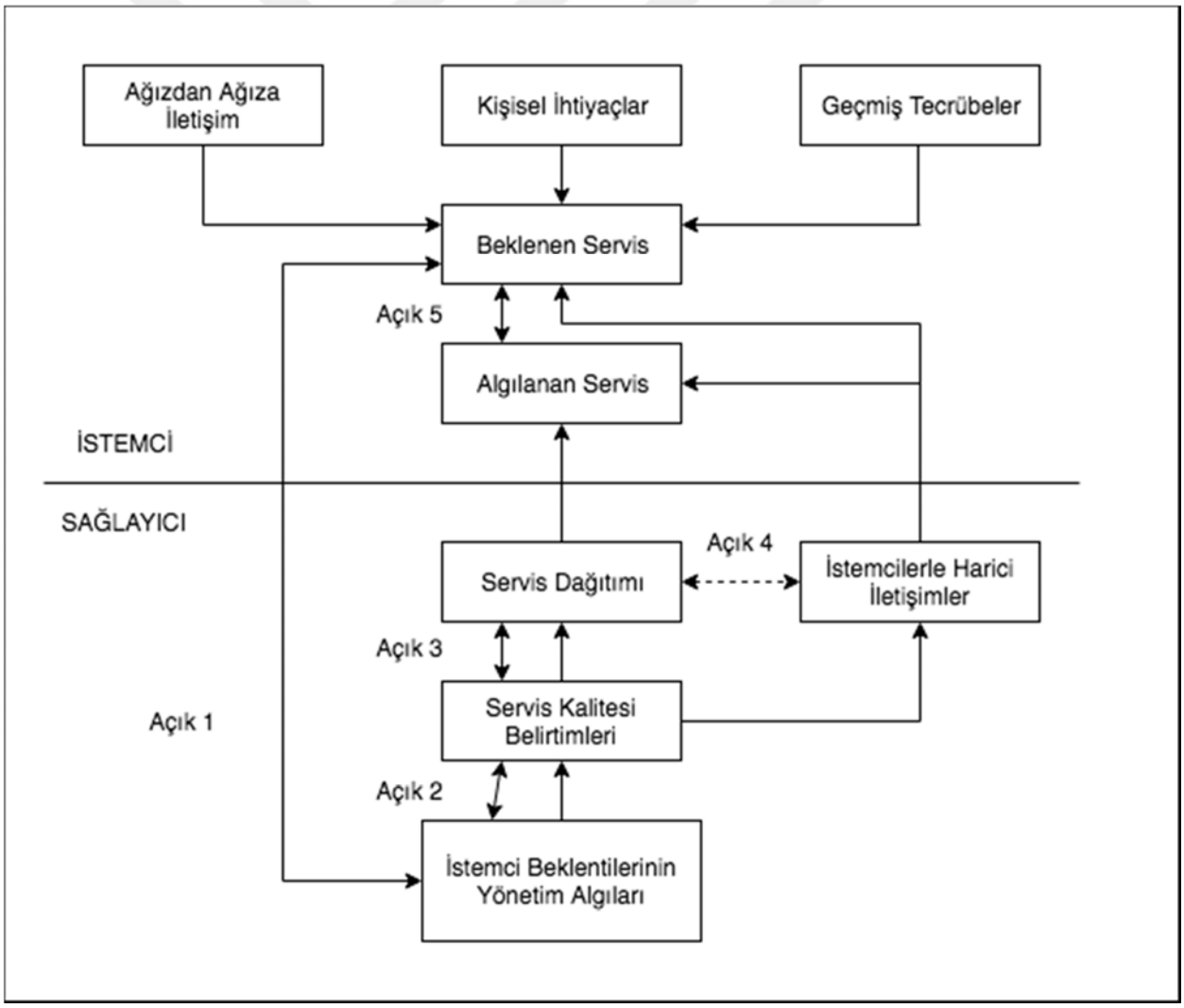
Güven ayarı: Güven ilişkisi, zaman, yer ve servislerin değişmesi veya aralarındaki güvensizliğin ortaya çıkması nedeniyle bazen düzenlemeler gerektirebilmektedir. Bu düzenlemelere güven ayarı adı verilmektedir.

Güvenin sürdürülebilirliği: Sürdürülebilir bir güven ilişkisi, web servislerinde stratejik güven yönetimi için önemli bir elemandır. Bir güven ilişkisi artık sürdürülemez hale gelirse, o ilişki bir süre sonra bitmeye mahkumdur.

2.1.3. Web Servislerinin Kalitesi

Servis odaklı mimariye dayalı web servisleri, endüstri ve akademik çevreler tarafından giderek daha fazla dikkat çeken, büyüyen bir dağınık bilgi işlem teknolojisidir. Web servislerinin kalitesi, gittikçe daha önemli hale gelmektedir [9]. Web servis kalitesini gösteren bir model, Şekil 2.8'de verilmiştir [10]. Bu modelde, beş boşluk türünün önemi bulunmaktadır. Bunlar:

(1) istemci beklenti yönetimi algı açığı (Açık 1),
(2) yönetim algısı – servis kalitesi belirleme açığı (Açık 2),
(3) servis kalitesi özellikleri – servis sunumu açığı (Açık 3),
(4) servis sunumu – dış iletişim açığı (Açık 4),
(5) beklenen servis – algılanan servis açığı (Açık 5)
şeklindedir.



Şekil 2.8: Servis kalitesi modeli

2.1.4. Web Servislerinin İş Süreçleri

Bir web servisinin iş süreci; bir işletmenin istenen işlevlerini yerine getirmek için belirli şartlar altında web servislerinin otomatik olarak seçilmesi, entegrasyonu ve çağrılması işlemlerinin ne şekilde yapıldığını ifade eden süreçtir. Bu süreç esnasındaki en zorlu görev, insan müdahalesi olmaksızın belirtilen gereksinimleri ve kısıtlamaları karşılayan bir web servisinin seçilmesidir. Optimum bir web servisi için, algoritma, ilk önce fonksiyonel bir eşleştirme gerçekleştirir. Bir servisin istenen kısıtlamaları yerine getirmemesi durumunda, konum, platform, yürütme hızı vb. özelliklerine bakılmadan, mevcut olan birçok servis, iş akışının parçalarını yürüten bir başka servise entegre edilebilir.

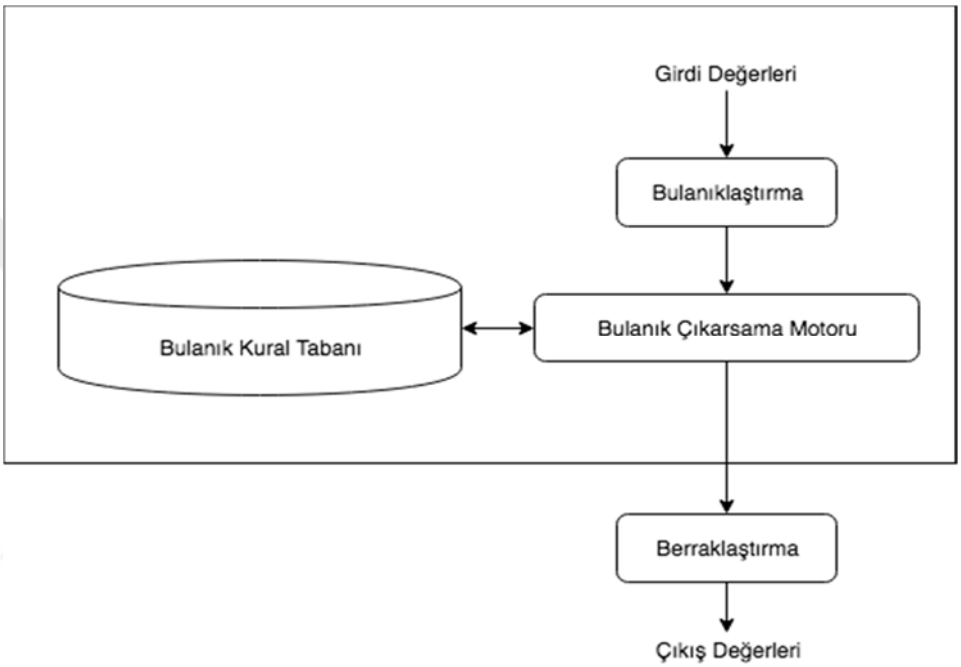
Son yıllarda pek çok farklı iş süreci modelleme notasyonları ortaya çıkmıştır. İş Süreçleri Modelleme Notasyonu (Business Process Modeling Notation – BPMN), İş Süreçleri Modelleme Dili (Business Process Modeling Language – BPML) ve Birleşik Modelleme Dili (Unified Modeling Language – UML) bunlardan bazılarıdır. Her dil, iş süreçlerini modellemek için farklı kavramlar, söz dizimi ve karmaşıklık sağlar. Web servis kompozisyonu, daha üst düzey iş süreçleri oluşturarak web servislerini birbirine bağlamak için açık, standartlara dayalı bir yaklaşım sunar [2].

2.1.5. Web Servislerinin Güvenilirliği

Günümüzde servis odaklı sistemler, pek çok sektörde uygulanmaktadır. Temel olarak bir servis topluluğu olarak ifade edilebilecek olan servis odaklı sistemlerin güvenilirliği, ilgili web servislerinin güvenilirliğine bağlıdır. Günümüzde, bir bilgisayar yazılımının güvenilirliğini tahmin etmek için, yazılım hata oranını model olarak gören bir dizi yöntem bulunmaktadır. Bu modellerin çoğunda yazılımdaki hata oranlarının deterministik bir fonksiyon olduğu varsayılır. Ancak servis odaklı sistemlerde, güvenilirlik web servislerine bağlıdır ve farklı kullanıcılar için aynı web servisi farklı hata oranları verebilir. Gizli Markov Modeli (Hidden Markov Model – HMM) ve bulanık mantık yöntemleri web servisinin güvenilirliğini tahmin etmek için kullanılabilir.

Bulanık mantık yöntemindeki bulanık değişkenler kesin değildir; bu da, kısmen bilinen ve doğrusal olmayan sistemler için verimli çözümler sunar. Üyelik fonksiyonları, girişler ve çıkışlar arasındaki ilişkileri çıkarmak için kullanılır. Genel olarak, bu üyelik işlevleri değişkenleri tanımlamak için 0 ile 1 arasında bir değer aralığı kullanır. Bu aralık, bulanık modeli klasik Doğru – Yanlış mantıksal konseptinden ayırır.

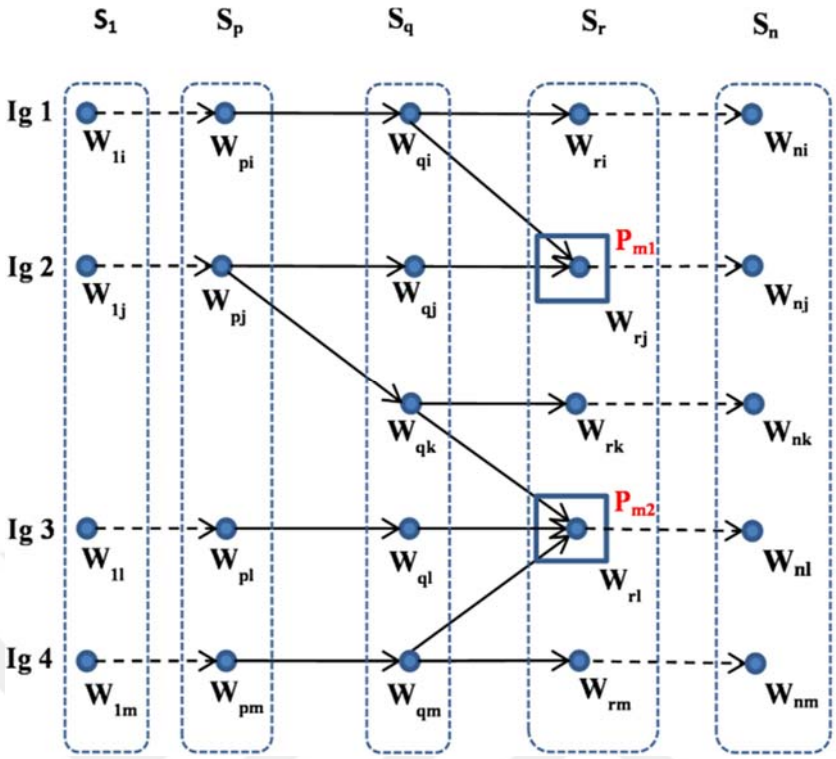
Bir bulanık sistem, temelde 4 bileşen içerir (Şekil 2.9). Bunlar, bulanıklaştırma, bulanık çıkarım motoru, bulanık kural tabanı ve berraklaştırmadır. Bulanık mantıkta ana fikir, evrensel kümenin farklı alt kümelerine kısmen ait olan bir nesneye izin vermesidir. Üyelik fonksiyonları, bu kısmî verileri 0 ile 1 aralığına dönüştürmek için kullanılır. Bulanık değişkenlere üyelik fonksiyonları atamak için, genetik algoritmalar, yapay sinir ağları, sezgisel yöntemler ve derece sıralaması gibi farklı yaklaşımlar kullanılır [11].



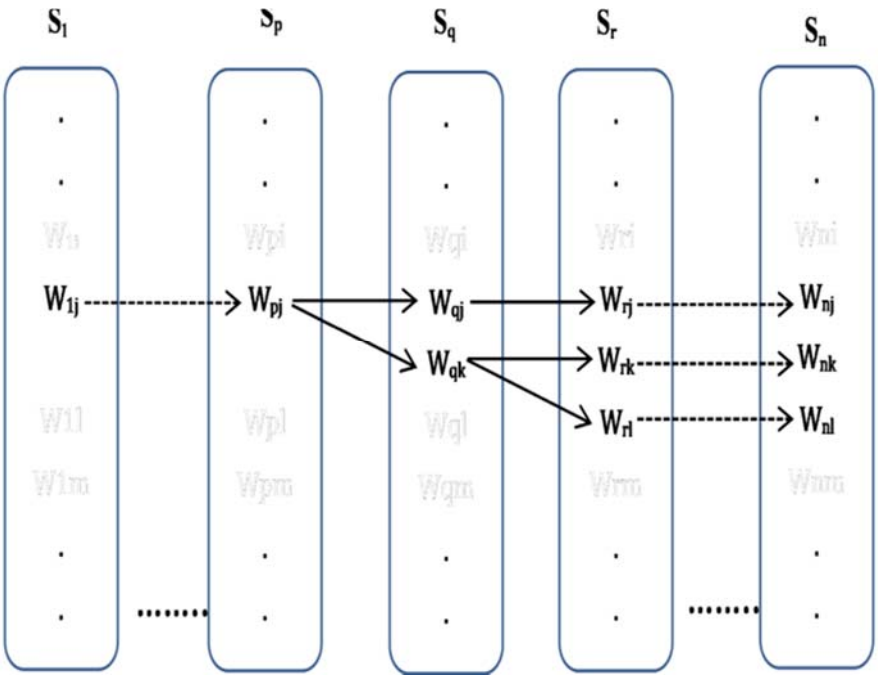
Şekil 2.9: Bulanık mantık modelleme süreci

2.1.6. Web Servislerinin Bileşimi

Web servislerinin bileşimi, bir servisin diğer uzaktan servislerle bir araya getirilmesi, katılımcı olan servislerin belirtilmesi, servislerin çağırma sırası ve istisnaların işlenmesine gibi işlemlerin bütünüdür. Web servisleri bileşimlerinin tasarlanması ve çalıştırılması hataya açıktır çünkü tek bir servis, görevlerini yerine getirmek için diğer servislere bağlı olabilir. Web servisi bileşimlerinin yürütme sırasında nasıl davrandığını ve işlevsel gereksinimlere uyup uymadıklarını tahmin etmek zordur. Etkileşim, eşzamanlılık ile ilgili sorunlara yol açar. Bu eşzamanlılık hatalarının test ile elde edilmesi zordur, çünkü bunlar "tekrar üretilemez olma" veya "test vakaları tarafından kapsanma" eğilimindedir. Şekil 2.10'da bir web servis bileşimi senaryosunun konsolide edilmiş görünümü, Şekil 2.11'de ise bileşim zincir ateşleyicisinin ayırt edilebilir görünümü verilmiştir.



Şekil 2.10: Bir web servis bileşim senaryosunun konsolide edilmiş görünümü



Şekil 2.11: Bileşim zinciri ateşleyicisinin ayırt edilebilir görünümü

2.1.7. Web Servislerinin Seçimi

Web servisi mimarisi, atomik iş süreçleri arasında birlikte işlerlik elde etmeyi mümkün kılar. Günümüzde birçok uygulama tarafından karmaşık bileşik web servislerin oluşturulması sağlanmıştır. Bir kompozit web servisi oluşturulması, bir atomik servis ile ilgili işlevsel olan ve işlevsel olmayan iki temel gereksinim ihtiyacının araştırılmasını gerektirebilir: çeşitli web servislerin aynı işlevsel görevi gerçekleştirdiği bir ortamda, seçim yapmak oldukça önemli bir husus haline gelmektedir. Servisin "itibarı", birleşim için servisleri seçerken dikkate alınması gereken önemli bir parametredir. Kullanılabilirlik ve güvenilirlik hesaplama itibarı için ayrı ayrı alındığında, bir servisin çalışır durumda olma olasılığını artıracaktır ya da bu servisin hata eğilimindeki değişmeyi analiz edecektir [12].

2.1.8. Otonom Web Servisleri

Servis odaklı bilgi işlem paradigmasının giderek benimsenmesiyle, web servisinin kendi kendine adaptasyonu için etkin ve etkili mekanizmalara olan ihtiyaç giderek artmaktadır. Ek olarak, servis sağlayıcılar ve servis kayıtları gibi elemanların, öz-yönetim sürecinde kritik ve özel bir role sahip oldukları için kendilerine daha fazla adapte olmaları gerekmektedir. Etkin bir öz-yönetim için, otonom bir web servisi yeteneklerinin zengin bir açıklamasına sahip olmalıdır. Hizmet Kalitesi (Quality of Service – QoS) verileri öz-adaptasyon sürecini etkin bir şekilde sürmek için her zaman yeterli değildir. Bu verilere ek olarak, otonom web servislerinin belirli uyarlama protokolleri gibi ek bilgilerle belirtilmesi gerekmektedir. QoS tabanlı bir özyönetim sisteminde, QoS özniteliklerinin ve servisle ilgili verilerin değerlerini temsil etmek ve saklamak, kalite verilerinin otonom yöneticiler tarafından kullanılan temel kaynaklar olması nedeniyle temel bir sorundur [13].

2.1.9. Web Servisi İşlemleri

Web Servisi işlemlerinin temel ilkesi, web servislerinin güvenilir şekilde yürütülmesini ve genellikle aynı anda erişilen temel verilerin tutarlılığını sağlamaktır. Bu sebeple, literatürde web servis işlemlerinin güvenilirliğini ve verimliliğini arttırmak için çeşitli çözümler önerilmiştir. Bir web servisi işlemi, bir sonuca ulaşmak olan bir faaliyet akışı tarafından gerçekleştirilen mantıksal bir iş birimidir. Bir yürütücü, bir faaliyeti yürütmekten sorumlu bir web servisidir. Bir yürütücü, aşağıdaki yaygın olarak kullanılan durumların herhangi birinde olabilir: "Başlangıç, Etkin, Tamamlandı, Dengelendi, Durduruldu, İptal edildi ve Başarısız". Bir yürütücünün durumu, ilkel bir eylemin yürütülmesiyle değiştirilir. Altı atomik ilkel eylem

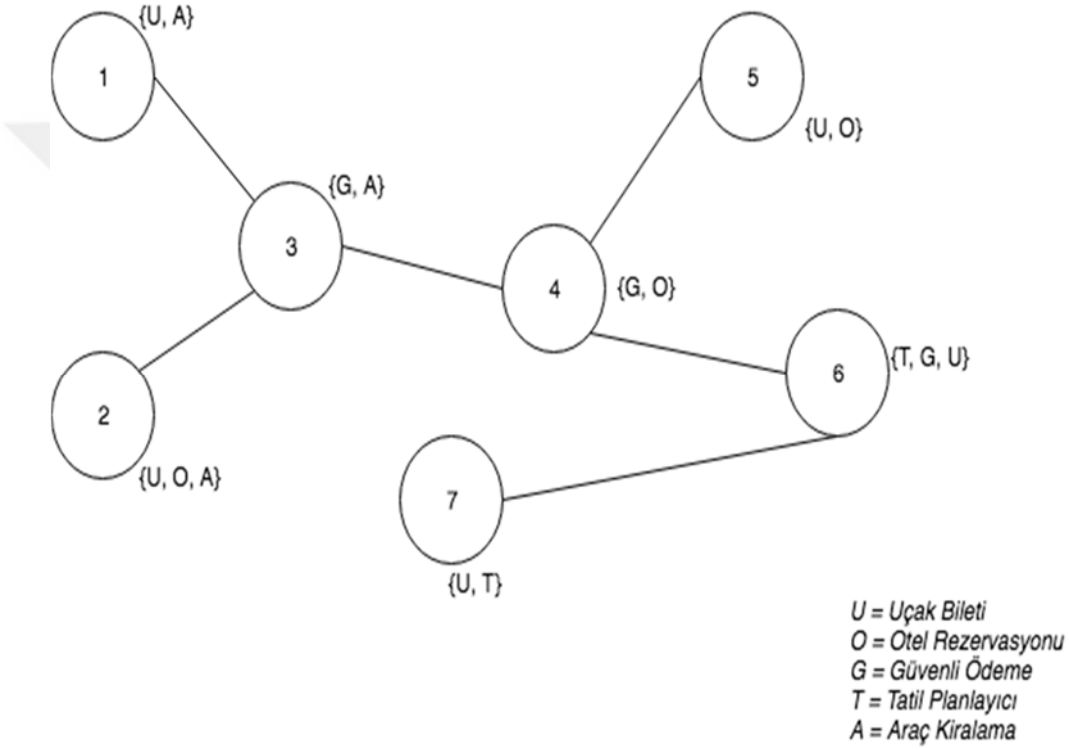
vardır: “Başlangıç, Tamamlama, Telafi Etme, Geri Çekme, İptal ve Başarısız Olma”. İlkel eylem telafisi yalnızca faaliyetin telafi edilebilir olması durumunda uygulanabilir. Bir yürütücü, işleme ilk kaydedildiğinde ve çalıştırılmayı beklerken Başlangıç durumundadır. Bir yürütücü, ilkel eylemi başlattığında ve yürütmeyi tamamlamadığında Etkin durumdadır. Etkinliğini başarıyla tamamladıktan sonra Tamamlandı durumuna geçer. Tamamlanan durumdan, etkinlik telafi edilebiliyorsa, yürütücü Dengelendi durumuna girebilir. Bir yürütme, iptal ilkel eylemlerden birini gerçekleştirdikten sonra Durduruldu durumuna gelir. Eğer yürütücünün, faaliyetini yürütürken iptal edilmesi durumu söz konusu olursa, yürütücü İptal durumuna geçmiş olur. Son olarak, bir yürütücü bir etkinliği, etkinliğini başarılı bir şekilde bitiremediyse Başarısız durumuna geçer.

Web servislerindeki testin amacı, bir sistemin veya bir bileşenin beklenmedik davranışını sistematik olarak araştırmaktır. İdeal olarak, bir yazılımda, Test Altındaki Yazılım’ın (Software Under Test – SUT) olası tüm durumları test edilmelidir. Fakat teorikte bunun mümkün olmadığı kabul edilmektedir çünkü bir yazılımdaki girişlerin tüm olası kombinasyonların sayısı sonsuz olabilir. Ayrıca, test süreci zaman, para ve işgücü gibi kaynaklara ihtiyaç duyar. Bu nedenlerden dolayı, testin etkililik/maliyet dengesini dikkate alarak gerçekleştirildiğinden emin olmak için test teknikleri kullanılır. Test teknikleri, yazılım ya da test edilen eleman hakkında bazı bilgileri kullanarak tasarım durumları tasarlamak için rehberlik sağlar. Test için en uygun koşulları ve her koşul için en önemli değerleri sistematik olarak belirlemelerine izin verir. Bir testin temeli, bir bileşenin veya sistemin gerekliliklerinin çıkarılabileceği tüm kaynaklardır. Bunlar, test parçalarına ayrılır ve her parça için bir dizi test koşulu türetilir. Bir test koşulu, bir veya daha fazla test vakası tarafından doğrulanabilen bir bileşen veya sistemin bir öğesi veya olayıdır. Test koşullarının uygulanması sonucunda eleman test edilmiş olur [14].

2.1.10. Web Servislerinin Sosyal Ağı

Internet üzerinde, günümüzde çok sayıda web servisi bulunmaktadır ve bunlar arasında doğru olanları tanımlamak, her zaman basit bir görev değildir. Ayrıca, bağımsız sağlayıcılar, farklı işlevsel olmayan özelliklerle neredeyse aynı web servisleri kümesini sunmaya devam ederler ki; bu da, keşif sürecinin karmaşıklığını artırıcı bir unsurdur. Bununla birlikte, bu web servisleri, tam donanımlı bir sistem veya ürünün oluşması için ya da daha büyük web servisleri oluşturmak için birbirleriyle sürekli olarak etkileşime girerler. Web servisleri arasındaki etkileşimler, bir sosyal ağ servisleri ağına giden bir ilişkiler ağı oluşturur.

Bir web servisleri sosyal ağı, düğümleri ve düğümler arasında çeşitli bağlantıları olan bir çizge olarak görülebilir. İki web servisi arasındaki yönsüz bir kenar, potansiyel olarak birbirleriyle iletişim kurabilecekleri anlamına gelir. Kenarın ağırlığı, iletişim maliyetini (gecikmeyi) temsil eder. Sosyal ağdaki her web servisinin bir veya birden fazla işlevi vardır. Her fonksiyon, bu fonksiyonun geçmiş performans kalitesini yansıtan bir puan ile etiketlenir. Bu puan, düşük, orta seviyeli veya yüksek olabilir. Şekil 2.12'de, örnek bir web servis sosyal ağı çizgesi görülmektedir.



Şekil 2.12: Web servislerinden oluşan sosyal ağ örneği

2.2. WEB SERVİS ÇEŞİTLERİ

Bu bölümde en yaygın kullanılan web servis çeşitleri ve özellikleri açıklanmıştır.

2.2.1. SOAP Web Servisleri

SOAP (Simple Object Access Protocol – Basit Nesne Erişim Protokolü), dağıtılmış bir ortamda eşler arasında yapılandırılmış ve yazılan bilgileri değiştirmek için basit ve hafif bir mekanizma sağlayan bir protokoldür. SOAP, programlama modeli veya uygulamaya özgü herhangi bir uygulama semantiğini kendisi tanımlamaz; modüler bir paketleme modeli ve modüller içindeki verileri kodlamak için şifreleme mekanizmaları sağlar. SOAP üç bölümden oluşur:

- SOAP zarfı: Bir mesajın içeriğini ifade etmek için genel bir çerçeve tanımlar. “Bu mesajla kim ilgilenmeli?” ve “Bu mesaj isteğe bağlı mı yoksa zorunlu mu?” gibi sorulara cevap arayan bir elemandır.
- SOAP şifreleme kuralları: Uygulama tanımlı veri tiplerinin örneklerini değiştirmek için kullanılabilecek bir mekanizmadır.
- SOAP RPC gösterimi: Uzaktan prosedür çağrılarını ve cevaplarını temsil etmek için kullanılabilecek bir sözleşmedir.

Bu parçalar SOAP'ın bir parçası olarak birlikte tanımlanmış olsa da, işlevsel olarak ayrıktır. Özellikle, zarf ve şifreleme kuralları, modülerlik yoluyla basitliği arttırmak için farklı ad alanlarında tanımlanmıştır.

SOAP mesajları temel olarak bir göndericiden alıcıya tek yönlü iletimlerdir, ancak yukarıda gösterildiği gibi, SOAP mesajları genellikle istek/cevap gibi kalıpları uygulamak için birleştirilir. SOAP uygulamaları, belirli ağ sistemlerinin benzersiz özelliklerinden yararlanmak için optimize edilebilir. SOAP'ın bağlı olduğu protokolden bağımsız olarak, mesajlar, nihai hedefe ek olarak bir veya daha fazla ara düğümde işlenmeye izin veren "mesaj yolu" olarak adlandırılan bir yol boyunca yönlendirilir. SOAP mesajı alan bir SOAP uygulaması, aşağıdaki eylemleri aşağıdaki sıraya göre yerine getirerek işleyecektir:

1. Bu uygulamaya yönelik SOAP mesajının tüm bölümlerini tanımlar.
2. 1. adımda belirtilen tüm zorunlu parçaların bu mesajın uygulaması tarafından desteklendiğini doğrular ve bunları uygun şekilde işler. Durum böyle değilse, mesajı atar. İşlemci, işlemin sonucunu etkilemeden 1. adımda tanımlanan isteğe bağlı parçaları göz ardı edebilir.
3. SOAP uygulaması mesajın nihai hedefi değilse, mesajı iletmeden önce 1. adımda tanımlanan tüm parçaları kaldırılır.

Bir mesajın veya mesajın bir parçasının işlenmesi, SOAP işlemcisinin, diğer şeylerin yanı sıra, kullanılan alış biçimi (tek yönlü, istek/cevap, çok noktaya yayın vb.), bu modeldeki alıcının rolü istihdam durumunu anlamasını gerektirir.

2.2.2. Rest Web Servisleri

REST (REpresentational State Transfer – Temsili Durum Transferi), gösterimi yapılan durum transferi anlamına gelir. Sistemin, basit bir web tabanlı sosyal uygulama üzerinden açıklaması şu şekildedir:

1. Bir kullanıcı, adresi tarayıcıya yazarak uygulamanın ana sayfasını ziyaret eder.
2. Tarayıcı, sunucuya bir HTTP isteği gönderir.
3. Sunucu, bazı bağlantılar ve formlar içeren bir HTML belgesiyle cevap verir.
4. Kullanıcı bir forma bilgilerini yazar ve formu gönderir.
5. Tarayıcı, sunucuya başka bir HTTP isteği gönderir.
6. Sunucu isteği işler ve başka bir sayfa ile cevap verir.

Birkaç istisna dışında, çoğu web sitesi ve web tabanlı uygulama aynı modeli takip ettiği bu döngü, kullanıcı durdurana kadar devam eder. Bu konuyla ilgili önemli bazı kavramlar aşağıda açıklanmıştır:

Tekdüzen Kaynak Konum Belirleyicisi: Önceki etkileşimin başlangıcında kullanıcının tarayıcıya yazdığı şey, Tekdüzen Kaynak Konum Belirleyicisidir (Uniform Resource Locator – URL).

Kaynak: Bir kaynak, bir URL tarafından tanımlanabilen herhangi bir şeydir. Tipik bir statik web sitesinde, her web sayfası bir kaynaktır.

Tekdüzen Arayüz: İstemciler ve kaynaklar arasındaki tüm etkileşimler, birkaç temel HTTP metodu aracılığıyla gerçekleştirilir. Herhangi bir kaynak, bu metodların bazılarını veya tümünü açığa çıkarır ve bir metot, onu destekleyen her kaynakta aynı şeyi yapar. İstemci, tanımlı HTTP yöntemleri ile etkileşim kurar. Bu yöntemler GET, POST, PUT, DELETE metodlarıdır. Bu yöntemler, bir kaynağa neler yapabileceğini tanımlar.

HTTP GET: Bu metod, Talep – URI tarafından tanımlanan herhangi bir bilgiyi (bir varlık biçiminde) tanımak anlamına gelir. RESTful web servislerinde, URI tarafından belirlenen bu bilgi, bir kaynağın gösterimidir. GET güvenli ve eş kuvvetli bir metoddur. Güvenli olması sunucunun durumunu değiştirmediği anlamına gelir, sadece bilgileri alır, böylece herhangi bir yan etkisi olmaz. Eş kuvvetli, tek bir istek ile aynı etkiye sahip birden fazla özdeş istek uygulanabileceği anlamına gelir. HTTP protokolü durumsuz bir protokol olduğu için, güvenli

olarak tanımlanabilmesi için de eş kuvvetli olmalıdır. Eş kuvvetli olmak, hata durumunda GET isteklerini güvenli bir şekilde tekrar etmeyi sağlar. İstek başarılı olursa ve teslim alan kaynak mesaj gövdesinde döndürürse, uygun cevap durumu 200'dür (TAMAM). Kaynak yoksa, cevap durumu 404'tür (BULUNAMADI).

HTTP POST: Bu metod, istekte bulunan veriler ile verilen URI'nin yeni bir alt sıfatı olarak yeni bir kaynağın oluşturulması gerektiğini talep etmek için kullanılır. Gerçekleştirilen eylem, bir URI tarafından tanımlanabilen bir kaynak olarak sonuçlanmazsa, uygun cevap durumu 200'dür (TAMAM). Bir kaynak oluşturulduysa, cevap 201 (OLUŞTURULDU) olmalı ve isteğin durumunu tanımlayan ve yeni kaynağa ve bir konum başlığına işaret eden bir varlığı içermelidir.

HTTP PUT: PUT metodu, tutulan bir varlığın sağlanan adres altında saklanmasını talep eder. URI zaten varolan bir kaynağa başvurursa, tutulan varlık sunucudaki değiştirilmiş bir sürüm olarak düşünülmelidir. Mevcut bir kaynağa işaret etmiyorsa ve bu URI istekte bulunan kullanıcı aracısı tarafından yeni bir kaynak olarak tanımlanabiliyorsa, kaynak sunucu bu URI ile kaynağı oluşturabilir. Yeni bir kaynak oluşturulduysa, cevap durumu 201 (OLUŞTURULDU) olur. Mevcut bir kaynak değiştirildiğinde, talebin başarılı bir şekilde tamamlandığını gösteren 200 (TAMAM) cevap kodu gönderilmelidir. Kaynak yaratılamadıysa veya değiştirilemezse, sorunun niteliğini yansıtan uygun bir hata cevabı verilmelidir.

POST ve PUT arasındaki ana fark, isteğin farklı anlamlarda yansıtılmasıdır. Bir PUT isteğindeki URI, sorgu ile tutulan varlığı tanımlar. Tersine, bir POST isteğindeki URI, tutulan varlığı ele alacak kaynağı tanımlar. Dahası, belirli bir URI'ye birden fazla PUT talebi yapmanın, sadece bir tane yapmakla aynı etkiye sahip olması gerektiğinden, PUT yöntemi, GET yönteminin yanı sıra eş kuvvetlidir, ancak POST yöntemi değildir.

HTTP DELETE: DELETE metodu, sunucunun istek tarafından tanımlanan kaynağı silmesini ister. Ayrıca PUT ve GET metodları gibi eş kuvvetli bir yöntemdir. İşlem gerçekleştirildiyse, cevap başarılı bir 200 (TAMAM) olmalıdır, işlem gerçekleştirilemezse, sorunun niteliğini yansıtan uygun bir hata cevabı verilmelidir.

Gösterim: Bir gösterim, kaynağın XML, JSON veya HTML gibi bir format kullanılarak şifrelenmiş bilgilerinin (durum, veri veya işaret) kapsüllenmesidir. Sunucunun istemciye döndüğü HTML belgesi, kaynağın bir gösterimidir. Bir kaynak bir veya daha fazla gösterime

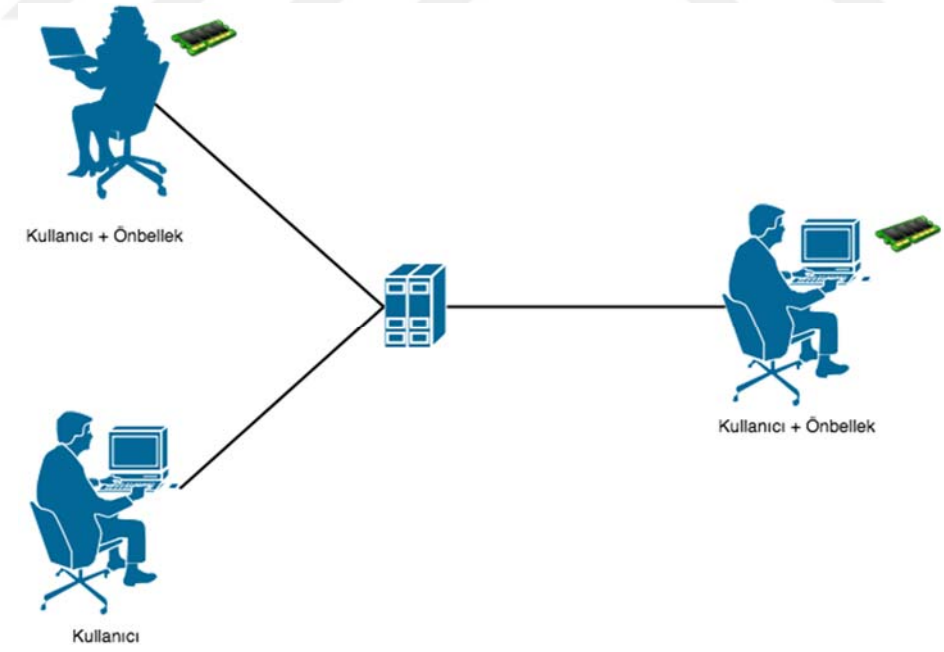
sahip olabilir. İstemciler ve sunucular, alıcı tarafa (istemci veya sunucu) gösterimin türünü belirtmek için ortam türlerini kullanır.

Hiper Ortam ve Uygulama Durumu: İstemcinin sunucudan aldığı her gösterim, kullanıcının uygulama içindeki etkileşim durumunu belirtir. Örneğin, kullanıcı başka bir sayfa almak için formu gönderdiğinde, kullanıcı uygulamanın durumunu kendi bakış açısıyla değiştirir. Bir kullanıcı sadece bir web sitesine göz attığında, kullanıcı başka bir sayfa yüklemek için her bağlantıya tıklandığında uygulamanın durumunu değiştirir.

Rest Web Servisleri İlkeleri

Bu bölümde bir REST web servisinin sahip olması gereken ilkeler kısaca açıklanmıştır:

Önbellek: İstemci – durum bilgisiz – sunucu tipine önbellek işlemlerinde sınırlamalar getirilmesi, ağ kullanımını daha verimli hale getirecektir. Bu sınırlamalar dahilinde iletilen isteklerin ve alınan cevapların önbelleğe alınıp alınmaması gerektiği, etiketlenerek ifade edilmelidir. Önbelleğe alınabilir şeklinde işaretlenen bir cevap bilgisi, tekdüzen istekler aracılığıyla birden fazla kere kullanılabilir (Şekil 2.13).



Şekil 2.13: Sunucu, kullanıcı ve önbellek ilişkisi

Önbellek işlemlerinde çeşitli sınırlamalar getirilebilir. Bunların en önemli potansiyel katkıları, veri işlemlerinde yaşanan cevap süresini düşürmek, önbellek üzerinde yapılan etkileşimleri

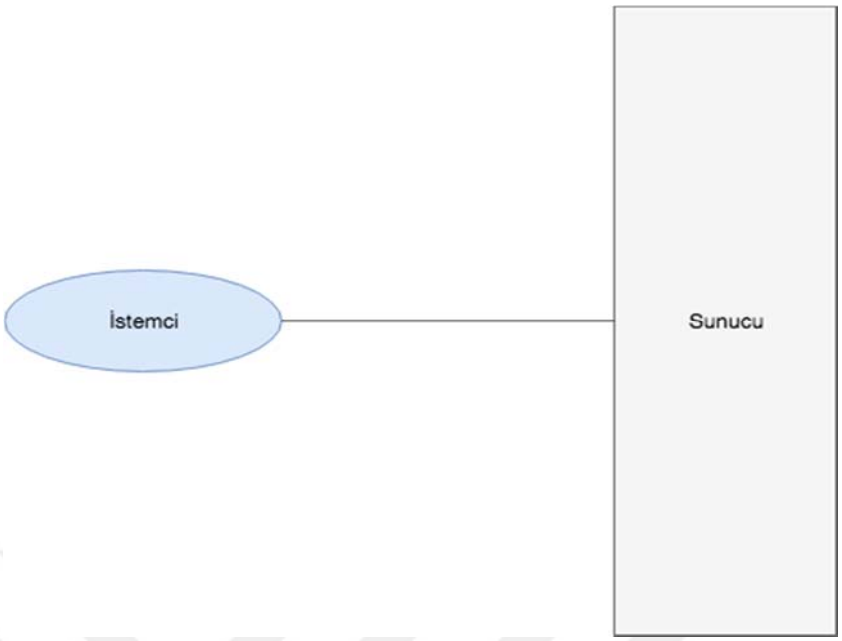
düzenleyerek daha verimli kılmak, ölçeklenebilirlik üzerine iyileştirmede bulunmak ve son kullanıcının kullanım hızını üst seviyeye taşımaktır. Yapılan sınırlamaya karşılık olarak, önbellekte tutulan bilgi en güncel veriden oluşmuyorsa, bu veri yine de kullanılmaya devam edebilir. Bu durum uygulamadaki veri akışının güvenilirliğini olumsuz yönde etkiler. HTML yapısıyla kurulan web siteleri de benzeri bir yapıya sahiptir ve çerez verisi aracılığıyla bu işlemi gerçekleştirir.

Zaman aşımı: Son kullanma, sunucunun talep edilen kaynağın ne kadar süreyle taze kaldığını söylemesi için bir yoldur. Çok nadiren (örneğin görsel dosyaları) veya belirli bir sürede değişebilen kaynaklar için mükemmeldir.

İstemci-sunucu: Dizayn olarak bulundurulması gereken sınırlamalardan biri istemci-sunucu (client-server) yapısıdır. İstemci-sunucu yapısının birincil özelliği, taraflar arasındaki bağlantıyı birbirinden ayrı tutmasıdır. Belirtilen taraflar, son kullanıcının işlemlerini gerçekleştirdiği arayüz ile bilginin tutulduğu veritabanı platformudur. Farklı işlemler için kullanılan yapıların ayrık tutulması, uygulamayı daha kolay yönetilebilir hale getirir ve ölçeklenebilirlik olarak ifade edilen kriterini iyileştirir. Sonuç olarak, farklı amaçlarla oluşturulan yapıların ayrı platformlarda geliştirilebilmesi web uygulamalarına da olumlu katkıda bulunur ve istemci – sunucu ilişkileri açısından daha geniş bir kapsamda kullanılabilme imkanı sunar [15].

İstemci-sunucu arabirimi, istekleri gönderen bir istemci bileşeninin ve istekleri alan bir sunucu bileşeninin varlığını gerektirir ve bir cevap verebilir (Şekil 2.14). Bu kısıtlama, ilgili durumların ayrılması protokolüne dayanmaktadır. Tek tip bir arayüz, istemcileri ve sunucu arayüzlerini ayırır.

İlgili durumların ayrılması, örneğin istemcilerin bir sunucu ilgisi olan veri depolama ile ilgili olmadığı ve sunucuların, kullanıcı arabirimi veya istemci ilgisi olan kullanıcı durumuyla ilgili olmadığı anlamına gelir. Bu, sunucu bileşenlerini basitleştirerek, çoklu platformlar ve ölçeklenebilirlik boyunca arabirimlerin taşınabilirliğini geliştirir. Ayrıca, istemci tarafındaki mantığı ve sunucu tarafı mantığının bağımsız gelişimini destekler, çünkü her bir bileşen, aralarındaki arayüz değişmediği sürece, ayrı olarak ikame edilebilir ve geliştirilebilir. İstemci-sunucu, belki de diğer kısıtlamaların tüm eserlerinden bahsetmesi ve bu kısıtlama üzerine inşa edilmesi gibi en temel kısıtlamadır.



Şekil 2.14: İstemci-sunucu yapısı

Kendinden tanımlayıcı mesajlar: Bileşenler arasındaki mesajlar sadece veri içermekle kalmaz, aynı zamanda mesajın içeriğini nasıl işleyeceğini açıklayan tanımlayıcı veriler de içerir. Tipik tanımlayıcı veriler, gösterimin temsil ettiği kaynağın URI'si veya gösterimin türünde, örneğin dosya formatı içerisindedir. Mesajlar ayrıca bir mesajın amacını tanımlayan kontrol verilerini de içerir. Bir istemci, bir kaynak üzerinde gerçekleştirmek istediği eylemi belirleyebilir (yeni bir kaynak oluşturmak isteyip istemediğini, var olan bir kaynağı güncelleyip güncellemeyeceğini vb.).

Uygulama Durumunun Motoru Olarak Hipermedya (Hypermedia as the Engine of Application State – HATEOAS): Bilinen bir başlangıç kaynağı tanımlayıcısı haricinde, istemci yalnızca alınan gösterimlerdeki sunucu tarafından sağlanan durum geçişleri arasında seçim yaparak durumları değiştirir. Bir web tarayıcısının kullanıcı arayüzü genellikle böyle çalışır; bir kullanıcı bilinen bir yer imine gider, ardından web sayfaları arasında gezinir ve web sunucusu tarafından sağlanan formları veya bağlantıları tıklayarak eylemleri gerçekleştirir [16].

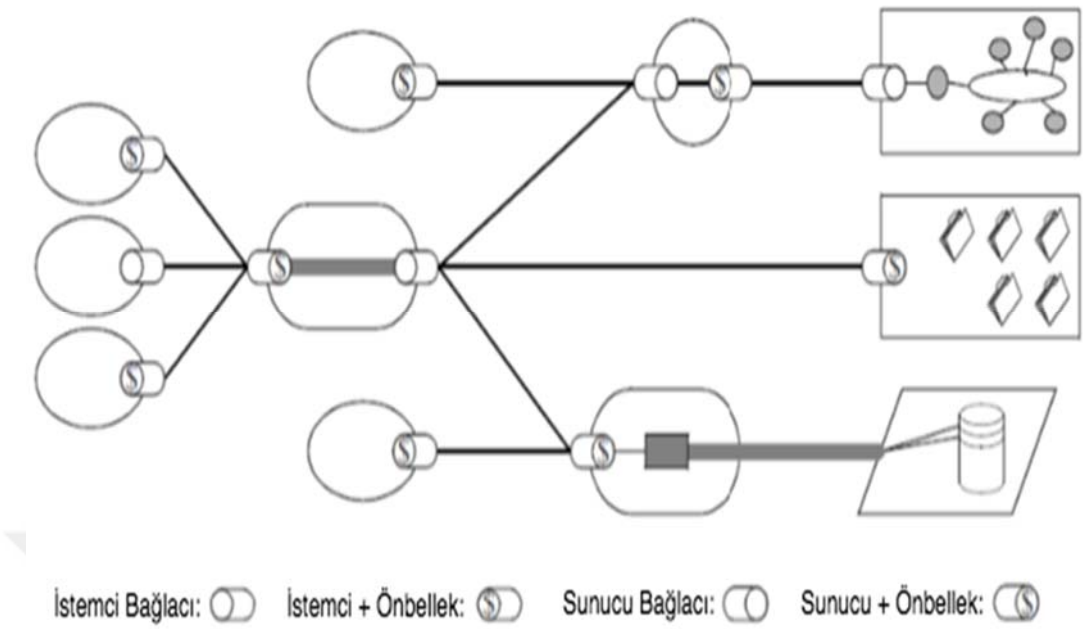
Katmanlı Sistem: Sistemlerin ayrıştırılarak kullanım yapısına göre katmanlara bölünmesi, internet uygulamalarının ihtiyaçlarına göre ölçeklendirilerek uzun vadede yazılım geliştirme süreçlerinin sürdürülebilir olmasını sağlar. Bu nedenle katmanlı sistem tavsiye edilmektedir. Her katman kendi içindeki işlemleri bulunduğu çerçevede kısıtlar ve sadece kendi içindeki

yapılarla iletişim halinde bulunabilir. Bunun dışında katmanlar bir önceki ve bir sonraki katman arasında iletişimde olabilir, daha uzak hiyerarşidekiler birbirlerine doğrudan erişemezler. Yapı itibariyle HTTP protokolü de bir transfer katmanından oluşmaktadır. HTTP protokolü de RESTful tipindeki web servisi tarafından kullanılmaktadır. Geliştirme süreci içerisinde, yeni servislerin eski servislerden ayırt edilmesine de katmanlar katkıda bulunur. Farklı yapıların aynı aracı birimi kullandığı durumlarda, sık kullanılmayan fonksiyonları ilerleterek işlemlerin daha yalın bir şekilde bileşenlerle harmanlanmasını kolaylaştırır. Bunun sonucunda aracı aynı zamanda ortak kullanılan bir aracı elemana nadiren kullanılan fonksiyonları kaydırarak bileşenlerin basitleştirilmesinde kolaylık sağlarlar. Bu sayede aracı birimler ağlar ile işlemcileri kapsayan servislerin yük dengesini düzenler. Sonuçta, yapıların ölçeklenebilirlik değerleri yükselir.

Sistem bileşenlerinin katmanlara ayrılmasında yaşanabilecek olumsuzluklar ise performansın azalmasına sebep olabilmesidir. Veri işlemleri sırasında darboğaz oluşabilir, bu da işlem süresini artırabilir. Sistem dizaynının kalitesine bağlı olarak bu olumsuzluğun önüne geçilebilir. Aracı elemanların kullanacağı önbellek, ağ üzerinden işlem yapan bir sistemin performansını düzenleyerek bu etkinin göz ardı edilmesini sağlar. İşlemlerin paylaşımlandığı bir önbellek yapısıyla bir düzenli veri alanı kurarak performans kazanımlarını bile sağlayabilir. Katmanlar arası geçişler olması, güvenlik duvarları için de bir bariyer görevi görür. Güvenlik protokollerinin uygulanması için uygun bir ortam sunar.

Katman uygulaması ve tekdüzen arayüz sınırlamalarının harmanlanması sayesinde, yazılım altyapılarının tasarımları “boru ve filtre” (pipe-and-filter) yapısıyla özdeşleşir. REST işlemleri iki yönlüdür fakat medya veri akışı ve büyük veri işlemleri tek yönlü işlem görebilir. Bilgi bir bütün halinde aktarılmalıdır, bu da ilgili filtrelerin sürece dahil edilmesiyle mümkün olur. REST sayesinde aracı birimler veri içeriğini dinamik olarak işleyebilir. Bu durum iletilen verinin tanım bilgisini içermesi sayesinde mümkün olmaktadır [15].

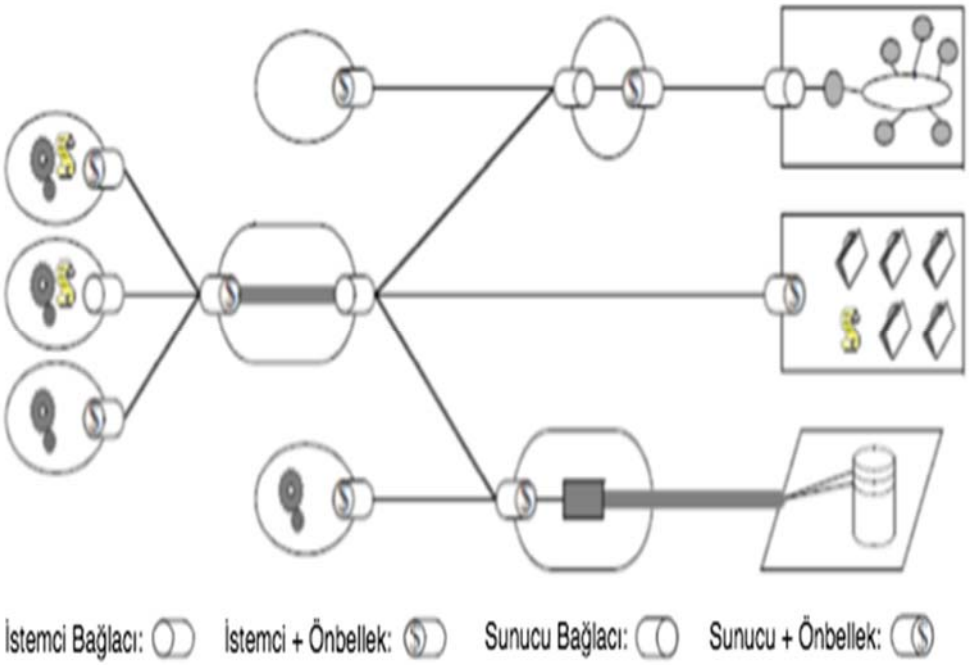
Katmanlı sistem stili, bir bileşenin, her bir bileşenin sistem hakkındaki bilgisinin, etkileşime girdiği anlık katmanla sınırlı olacak şekilde, bileşen davranışını kısıtlayarak, hiyerarşik katmanlardan oluşmasını sağlar (Şekil 2.15).



Şekil 2.15: Tekdüzen katmanlı istemci önbellek durum bilgisiz sunucu

Katmanlı bir sisteme sahip olarak, servislerin yalnızca ara katmanlarla iletişim kurabilmesini ve diğer katmanların onlar için görünmez olmasını sağlayarak güvenliği artırmayı mümkün kılar. Aracılar, çoklu ağlar ve işlemciler arasında servislerin yük dengelemesini sağlayarak sistem ölçeklendirilebilirliğini geliştirmek için de kullanılabilir. Katmanlı sistemlerin birincil dezavantajı, verilerin işlenmesine ek yük ve gecikme süresi eklemeleri ve kullanıcı tarafından algılanan performansı azaltmalarıdır. Bu dezavantaj, önbellek kısıtlaması gibi davranan ancak katmanlar arasında paylaşılan önbelleklerin kullanımıyla hafifletilebilir.

İhtiyaç Anında Kod: REST kümesinin son kısıtlaması, istemcilerin bir sunucudan kod yüklemesini ve çalıştırmasını sağlayan "isteğe bağlı" bir kısıtlama olan, İhtiyaç Anında Kod'dur (Şekil 2.16). İstemci işlevselliği, küçük uygulamalar veya komut dosyaları ile genişletilebilir. Bu durum, "ihtiyaç anında" olarak adlandırılır, çünkü uygulama ortamına bağlı olarak bazı avantaj ve dezavantajlara sahiptir ve içeriğe bağlı olduğu için, uygulamak her zaman mümkün değildir. Avantajları, istemcileri basitleştirmesi, böylece özelliklerin azaltılmış birleşimini teşvik etmesi ve sunucu yükleme-boşaltma çalışmaları sayesinde istemcilerin ölçeklenebilirliğini geliştirmesidir. Bununla birlikte, ihtiyaç anında kod, kodun kendisi tarafından üretilen görünürlüğü azaltır, bunu bir arabulucunun yorumlanması zordur [16].



Şekil 2.16: REST çalışma yapısı

Rest Web Servisleri Bileşenleri

REST bileşenlerinin fonksiyonları, Tablo 2.1'de gösterildiği gibi genel bir uygulama eyleminde sınıflandırılır.

Tablo 2.1: REST bileşenleri ve örnekleri

Kaynak Sunucu	Apache httpd, Microsoft IIS
Ağ Geçidi	Squid, CGI, Karşıt Sunucu
Vekil Sunucu	CERN Vekil Sunucu, Netscape Vekil Sunucu, Gauntlet
Kullanıcı Aracısı	Netscape Navigator, Lynx, MOMspider

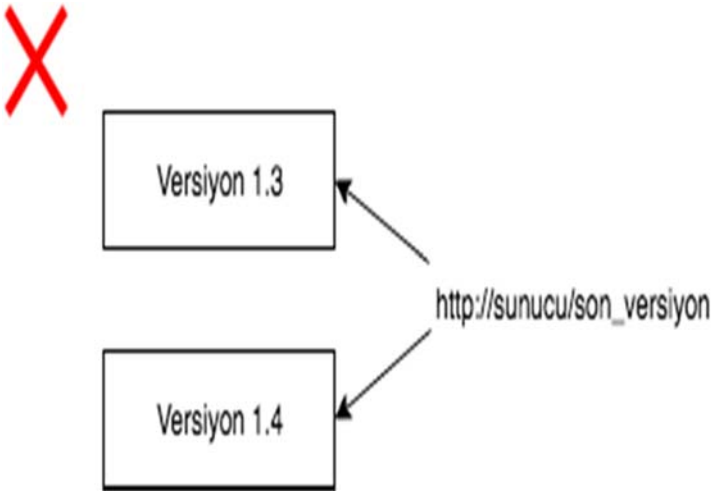
Bu tabloda adı geçen bazı elemanların açıklaması şu şekildedir:

1. Bir istemci aracısı, bir istek başlatmak ve cevabın nihai alıcısı olmak için bir istemci bağlayıcısı kullanır.
2. Bir başlangıç sunucusu, istenen bir kaynağın ad alanını yönetmek için bir sunucu bağlacı kullanır.

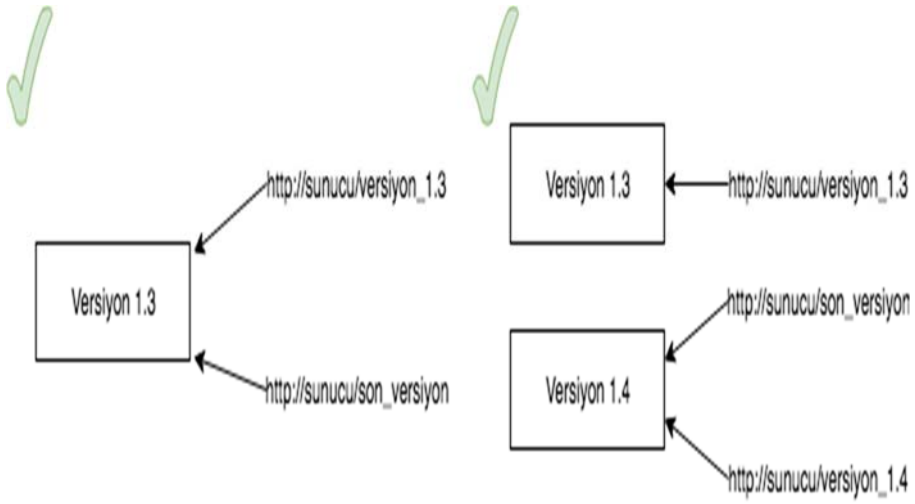
3. Bir arabulucu, diğer servislerin, veri aktarımının, performans artırımının veya güvenlik korumasının arayüz kapsüllemesini sağlamak için bir istemci tarafından seçilen bir aracıdır.
4. Bir ağ geçidi, diğer servislerin arabirim kapsüllemesi, veri çevirisi, performans geliştirmesi veya güvenlik uygulaması için, ağ veya başlangıç sunucusu tarafından uygulanan bir aracıdır [16].

Adreslenebilme: Kaynaklar adreslenebilir şekilde tanımlanmalıdır. Bu işlem URI'ler kullanılarak yapılabilir ve birebirdir. Kaynak sayısına eş sayıda URI tanımlanır. Sonuçta, her sayfaya işaret gösterilebilmektedir. URI'ler paylaşılabilir ve çok kere kullanılabilir. Bu sayede önbellekte tutulabilir. İlk defa alınan bir URI kopyalanır ve kaydedilir. URI için bir talep olması halinde bu kopya iletilir.

Kaynaklar: Kaynak, bir servisten edinilen bütün içerik olarak tanımlanabilir (kullanıcı bilgisi, medya içeriği, e-ticaret sitesi içeriği vb.). Kaynak, bilgi girişi yapan bir kişi veya bir servisin isteğinin arkasında tutulan bir veritabanı olabilir. URI başına bağlı birer kaynak bulunmalıdır, her biri kaynağa özgüdür. İstek edilen içeriğe has bir URI yoksa, bu kaynak sayılmaz ve web'de bulunamaz. Bununla birlikte bir URI iki farklı kaynak tarafından kullanılamaz. Ek olarak, iki farklı URI aynı kaynağa referans olabilir. İki kaynağın aynı URI'yi kullandığı durum Şekil 2.17'de, aynı veriye değinen iki farklı URI'nın olduğu durum ise Şekil 2.18'de gösterilmiştir.



Şekil 2.17: İki kaynak, aynı URI



Şekil 2.18: Aynı veriye değinen iki farklı URI

Tablo 2.2'de ise oluşabilecek metotlar ve açıklamaları verilmiştir.

Tablo 2.2: Temel HTTP cevap kodları

Kod	Metotlar	Açıklama
200	GET	Tamam - İstek başarılı: Gösterim istemciye gönderildi
201	POST, PUT	Oluşturuldu: Yeni kaynağın konumu, konum başlığındadır
201	PUT	Oluşturuldu: Kaynak zaten mevcutsa, içerik güncellendi
204	GET, PUT, DELETE	İçerik yok: İstek başarılı, istek gövdesi boş)
400	Tümü	Geçersiz istek: Hatalı oluşturulmuş istek
401	Tümü	Yetki dışı: İstemci yetkilendirilmelidir
404	Tümü	Bulunamadı: Kaynak sistemde bulunmamaktadır
405	Tümü	Metot için izin verilmiyor

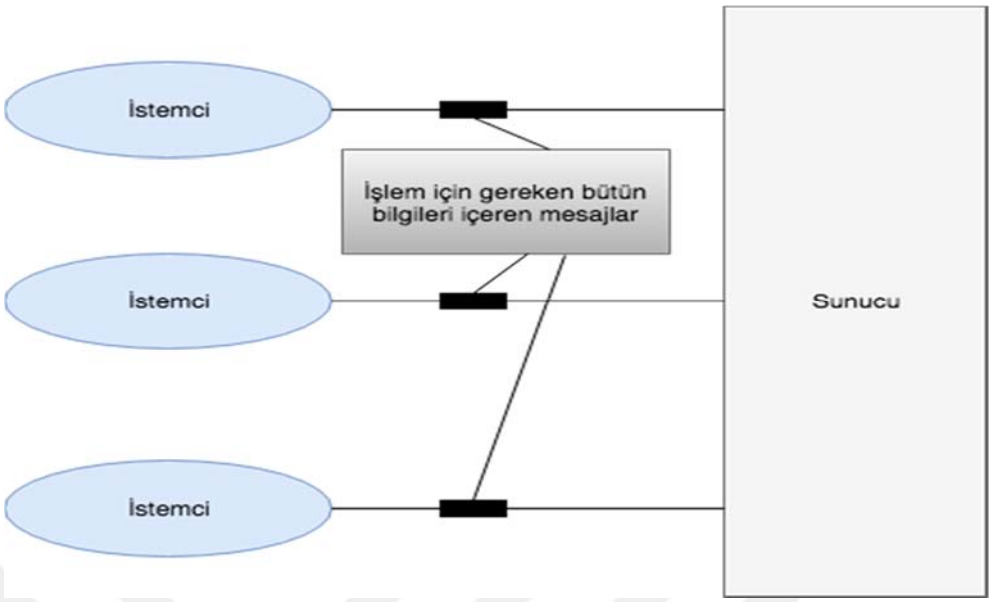
Durum ve durumsuzluk: Durum (state) bilgisi bulunan bir uygulamanın istemci-sunucu etkileşimine geçmemiş olması gerekir. Aksi takdirde durum bilgisine sahip olamaz, durumsuz olması gerekir. Bu tipte yapıya istemci-durum bilgisiz-sunucu adı verilir. Böyle bir uygulamada durum bilgisi bulunmadığından, istemciler beklemeye gerek duymadan sunucuya istek gönderebilirler ve gerekli cevap geri iletilebilir. İstemci de bu sayede oturum durum bilgisini kendi tarafında kaydedebilir, basit olarak adlandırılan web sayfalarında da bu şekilde kullanım yapılmaktadır.

İstemci - durum bilgisiz - sunucu sınırlaması sayesinde sistem daha güvenilir hale gelir ve daha ölçekli bir şekilde yönetilebilir. Sistemi izleyen bir yapı sadece istek içindeki veriyi inceleyerek içeriği belirleyebilir, bu da sistemin görünebilirliğine olumlu katkı sağlar. Hata ayıklama

kolaylaşacağından, bu istemci-sunucu yapısındaki sistem daha güvenilir hale gelir. Her bir istekten bir sonraki isteğe geçiş sırasında, her bir istek için durum bilgisi bulunmayacağından, bu içeriğe kaynak ayrılması gerekmez. Böyle bir ihtiyaç olmadığından daha yalın yapıda uygulamalar geliştirilir. Bunun da sistemin ölçeklenebilirliğine büyük bir katkısı olur. Mühendisliğin her alanında olduğu gibi, bu dizayn yapısında da bazı avantajların yanında dezavantajlar da bulunmaktadır. İstekler art arda hızlı bir şekilde iletilirken, gönderilen veriler tekrara uğrayabilir ve bu durum performansta düşüşe neden olur. Uygulama durum bilgisi istemcide saklanacağından, sunucunun uygulama üzerine kontrolü zayıflar [15].

İstemciler ve sunucular arasındaki iletişimin niteliği durumsuz olmalıdır. İstemci durumu sunucuda saklanmamalıdır, bunun yerine istemciler bir isteği işlemek için gereken tüm bilgileri dahil etmelidir. Sunucular, istekleri arasında herhangi bir durumu saklamak zorunda olmadıklarından, bu kısıtlama, sunucu ölçeklenebilirliğini ve sadeliğini artırır. Bu kısıtlamayla ilgili bir dezavantaj, istemciler daha fazla tekrarlayıcı veri göndermek zorunda kalabileceği için, bileşenler arasındaki iletişim performansını düşürebilmesidir. Ayrıca tutarlı uygulama davranışını sağlamak için istemcilere daha fazla sorumluluk yükler [17] (Şekil 2.19).

Durumsuzluk kısıtlaması: İstemci-Sunucu kısıtlamasına eklenir. Her etkileşimde, istemci ve sunucu arasındaki iletişim durumsuz olmalıdır. Bu, herhangi bir istemciden gelen her isteğin, sunucunun isteğin anlamını anlamasını sağlamak için gerekli olan tüm bilgileri içermesi gerektiği anlamına gelir (Şekil 2.19). Daha sonra, oturum durumuyla ilgili tüm veriler müşteriye iade edilmelidir. Bu nedenle, oturum durumu tamamen istemcide tutulur ve sunucu önceki isteklerden gelen bilgileri yeniden kullanamaz. Bu kısıtlama, bazı çok avantajlı özellikler ekler, ancak bazı dezavantajları da beraberinde getirir. Bir yandan, durumsuzluk, görünürlük, güvenilirlik ve ölçeklenebilirlik katmaktadır. Görünürlük artar, çünkü talep tüm bilgiyi içerdiğinden, talebin tam niteliğini belirlemek için ve önceki istekleri izlemek için bir izleme sistemine ihtiyaç yoktur. Kısmi hatalardan kurtulmak çok daha kolay olduğu için güvenilirlik gelişmiştir. Ölçeklenebilirlik daha iyidir çünkü istekler arasındaki durum bilgilerini depolamak zorunda kalmaz, sunucu bileşeninin kaynakları hızlı bir şekilde serbest bırakmasını sağlar ve ayrıca sunucunun kaynak kullanımını isteklere göre yönetmek zorunda kalmaması nedeniyle uygulamayı basitleştirir [16].



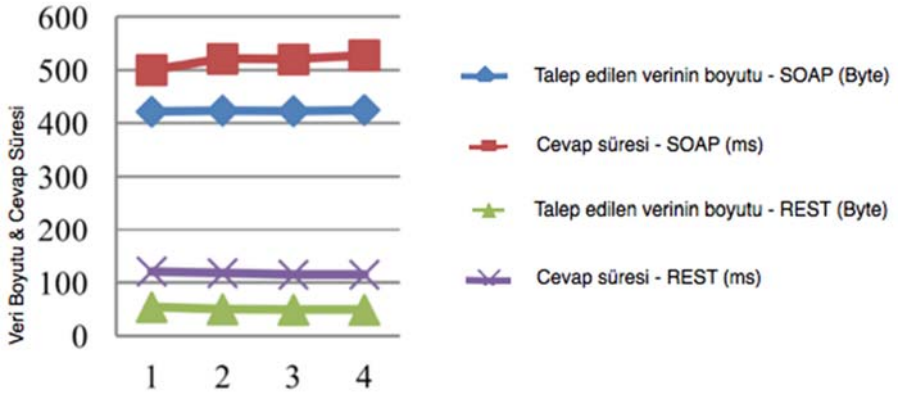
Şekil 2.19: İstemci – durum bilgisiz – sunucu yapısı

Rest ve SOAP Servisleri Karşılaştırmaları

SOAP, güvenlik ve servislerin oluşturulması açısından REST ile kıyaslandığında avantajlı durumdadır. SOAP servisleri veya SOAP istemcileri oluşturmak için çeşitli platformlar, Axis2/Java ve PHP SOAP gibi standart kütüphaneler sağlar. Uygulama, bir SOAP servisinin arabirimini tanımlayan WSDL'yi (Web Services Description Language – Web Servisi Açıklama Dili) gerektirir. Uygulanması için böyle bir API'ye sahip olmasından dolayı, hem sunucu hem de istemci, RESTful servislerini uygulamaktan daha kolay bir şekilde SOAP servislerini üretebilir.

Tablo 2.3: SOAP ve REST web servisleri için istenen verileri ve bunların milisaniye cinsinden cevap süresi

SOAP		REST	
İstenen verinin boyutu (Bayt)	Cevap süresi (ms)	İstenen verinin boyutu (Bayt)	Cevap süresi (ms)
422	500	54	121
424	521	50	118
423	520	49	115
425	528	49	115



Şekil 2.20: SOAP ile REST arasındaki talep edilen veri ile cevap süresi karşılaştırma grafiği

Şekil 2.20, SOAP ve REST web servislerinin cevap sürelerini gösterir. Sonuçlar SOAP web servisi ve REST web servisinden toplanır. Karşılaştırmadan sonra, REST web servislerinin verilere cevap vermek için daha az zaman aldığı açıktır [18].

2.2.3. Literatürdeki Çalışmalar

Bu bölümde, daha önce web servisleri ile ilgili olarak gerçekleştirilmiş çalışmalar özetlenmiştir.

Literatürde belli bir amaçla geliştirilen API'leri tanıtıcı çalışmalar mevcuttur. [19] çalışmasında modern işlemcilerin performans değerlendirmesi için kullanılan mobil bir API tanıtılmıştır. Bu çalışmanın amacı çoğu modern mikroişlemcide bulunan donanım performans sayaçlarına erişmek için standart bir uygulama programlama arayüzü belirlemektir. Bu sayaçlar, belirli sinyallerin ve işlemcinin işleviyle ilgili durumların meydana geldiği olayları sayan küçük bir kayıt kümesi olarak bulunur. Bu olayların izlenmesi, kaynak/nesne kodunun yapısı ile bu kodun altta yatan mimariye eşlenmesi verimliliği arasındaki korelasyonu kolaylaştırır. Bu korelasyonun; performans analizinde, elle ayarlama, derleyici optimizasyonu, hata ayıklama, kıyaslama, izleme ve performans modellemesi dahil olmak üzere çeşitli kullanımları vardır. Ek olarak, bu bilginin, yeni derleme teknolojisinin geliştirilmesinde ve mimari gelişimin yüksek performanslı bilişimde sık görülen tıkanıklıkların hafifletilmesine doğru yönlendirilmesinde faydalı olacağı düşünülmektedir.

[20] çalışmasında IP (Internet Protocol – İnternet Protokolü) ağ izleme için bir API tasarımı açıklanmıştır. Bu çalışma, ortaya çıkan uygulama ihtiyaçlarını karşılayacak kadar esnek ve mümkün olduğu durumlarda sistemin özel izleme donanımından yararlanmasına izin verecek genelleştirilmiş bir ağ akışı modeli üzerine inşa edilmiştir. Çalışmanın deneysel sonuçları, bu

çalışmada üretilen API'nin diğer yaklaşımlardan daha etkileyici bir güce sahip olduğunu ve aynı zamanda önemli performans iyileştirmeleri sağlayabildiğini göstermektedir.

[21] çalışması EnMAP veri işleme için bir araç ve API tasarlamıştır. Bu API, gelecekteki EnMAP verilerinin tam kullanımını garanti altına almak için iki amaç düşünülerek geliştirilmiştir: 1)EnMAP kullanıcı topluluğunu genişletmek, 2) görüntüleme spektroskopisi veri işleme için son yaklaşımlara erişim sağlamak. Çalışma, serbest bir şekilde temin edilebilir ve spektroskopik verilerin işlenmesi için klasik işleme araçlarının yanı sıra, komut dosyası dillerinde mevcut yöntemlerin entegrasyonu için güçlü makine öğrenme yaklaşımları veya arayüzleri de dahil olmak üzere, spektral görüntü işleme için çeşitli araçlar ve uygulamalar sunmaktadır. Özel bir geliştirici sürümü tam açık kaynak kodunu, bir uygulama programlama arayüzünü ve yeni gelişmelerin kolay entegrasyonu ve dokümantasyonu için bir uygulama sihirbazını içerir. Bu makale, kullanıcılar ve geliştiriciler için EnMAP'a genel bir bakış sunar, bir uygulama örneği boyunca tipik iş akışlarını açıklar ve görüntüleme spektroskopisi uygulamaları için sık kullanılan ve sürekli genişletilmiş bir platform haline getirme konseptini örneklendirmektedir.

Python programlama dili kullanılarak geliştirilen ve coğrafî kaynak analizleri için kullanılan bir API [22] çalışmasında açıklanmıştır. Bu çalışmada geliştirilen API, GRASS (Geographic Resources Analysis Support System – Coğrafi Kaynaklar Analiz Destek Sistemi) akademi, ticari ortamlar ve devlet kurumlarında yaygın olarak kullanılan güçlü bir açık kaynaklı GIS (Geographic Information System – Coğrafî Bilgi Sistemi) için nesne yönelimli bir Python API'sidir.

[23] çalışmasında bir C ++ sınıf kütüphanesi sağlayan bir API sunulmaktadır. Bu kütüphaneyi kullanarak uygulama sırasında uygulama programlarını enstrümanla değiştirmek mümkündür. Bu kütüphanenin benzersiz bir özelliği, makineden bağımsız ikili enstrümantasyon programlarının yazılmasına izin vermesidir. Çalışmada, bu kütüphaneyi kullanırken bir aracın gördüğü arayüz açıklanır. Ayrıca, bu arayüzü kullanarak oluşturulmuş üç basit araç da incelenmiştir: bir fonksiyonun çağrılma sayısını saymak için bir yardımcı program, halihazırda çalışan bir programın bir dosyaya çıktısını yakalamak için bir program ve koşullu kesme noktaları uygulaması.

[24] çalışmasında tanıtılan LibSBML, Sistem Biyolojisi Biçimlendirme Dili (Systems Biology Markup Language – SBML) biçiminde ifade edilen ve içeriği okumak, yazmak, işlemek ve doğrulamak için kullanılan bir uygulama programlama arayüzü kütüphanesidir. ISO C ve C ++ dilinde yazılmıştır, Common Lisp, Java, Python, Perl, MATLAB ve Octave için dil bağlantıları sağlar ve hem SBML'nin hem de kütüphanenin kullanımını ve kullanımını kolaylaştıran birçok özellik içerir. Geliştiriciler, kendi SBML ayrıştırma, manipülasyon ve doğrulama yazılımlarını uygulama çalışmalarından tasarruf ederek, uygulamalarına LibSBML'yi yerleştirebilirler.

[25] çalışmasında tanıtılan MLI, veri odaklı hesaplamaya dayalı dağıtılmış bir ortamda Makine Öğrenmesi algoritmaları oluşturmanın zorluklarını ele almak için tasarlanmış bir API'dir. Öncelikli hedefi, yüksek performanslı, ölçeklenebilir, dağıtılmış algoritmaların geliştirilmesini kolaylaştırmaktır. Sonuçları, mevcut sistemlere göre, bu arayüzün, en az karmaşıklık ve oldukça rekabetçi performans ve ölçeklenebilirlik ile çok çeşitli ortak Makine Öğrenmesi algoritmalarının dağıtılmış uygulamalarını oluşturmak için kullanılabileceğini göstermektedir.

Her yeni API tasarımı, özgün bir ürün olarak kabul edilme potansiyeline sahip olduğundan bu konu ile ilgili pek çok patent de bulunmaktadır. [26] patenti, bir SAN (Storage Area Network – Depolama Alan Ağı) heterojen bileşenlerini yönetmek için erişim seviyesindeki bir API'nin sistem ve yöntemini açıklamaktadır.

Mobil gelişmiş bir telefon API'sinin tasarımı ve kullanım şekli [27] patentinde açıklanmıştır. Bu patentte, merkezi kontrole sahip entegre bir kablosuz ve kablolu ağdan oluşan ve kablosuz ağın farklı protokolleri ve kablosuz ağın özelleştirilmiş servislerin sağlanabilmesi için kablolu ağların arasında geçiş yapmak üzere programlanmış bir arayüze sahip bir ürünü içermektedir. Uygulama, birçok ağ arasında kullanılabilir ve kablosuz ağdaki çeşitli uygulamalar için kablolu ağın mimarisinin ve kablosuz ağın ev konum kaydının kullanılmasını kolaylaştırmaktadır.

[28] patentinde yaygın sorgu çalışma zamanı sistemi ve konu ile ilgili bir API tasarlanmış ve açıklanmıştır. Bu çalışmada, bir sorgu çalışma zamanı mimarisi ve mimariye uygun örnek bir uygulama programlama arayüzü sunulmaktadır. Mimari, bir veya daha fazla XML (eXtensible Markup Language – Genişletilebilir İşaretleme Dili) sorgusu ve görünümünün gireceği ve sorguların ve görünümlerin farklı veri modellerinin birden fazla veri kaynağı üzerinden çalıştırılabileceği sorguların çevrilmesini sağlar. Mimari, giriş sorgularını ve görünümlerini,

ilgili sorgu veya görünümün anlamını temsil eden bir ara dil temsiline dönüştüren ön uç derleyiciler içermektedir.

Meta-model nesneleri için bir çatı (framework) ve bir API [29] patentinde ifade edilmiştir. Bu çalışmada, giriş bileşeni, verileri bir sınıf hiyerarşisi ile ilişkilendiren bir meta-veri modelini işler; burada sınıf hiyerarşisi, sınıf nesneleri, nitelikler, kurallar ve/veya davranışsal tanımlamalar arasındaki ilişki tanımlarını içerir. Başka bir özellik, meta-veri ek açıklamalarına sahip bir öğeyi almak için bir bileşen içeren bir veri yönetim sistemi içerir. Bir analiz bileşeni, çalışma zamanında meta veri ek açıklamalarının dağıtımı yoluyla öğe için bir yapı belirler. Sistem ayrıca meta veri sınıfı türevleri, meta veri sınıfları, meta veri bütünlüğü kuralları ve/veya meta veri sınıfı davranışını tanımlayan bir çerçeve bileşeni de içerebilir.

Bir sanal bellek sistemindeki fiziksel belleğin tahsisini kontrol eden bir API [30] patentinde tasarlanmıştır. Buradaki API, fiziksel belleğin sanal bellek sisteminde tahsisini kontrol etmek için çok görevli bir çalışma ortamındaki uygulama programlarına olanak sağlar. Ayrıca işlevi uygulamaların kod ve veriler için yumuşak bir sayfa kilidi belirlemesine olanak tanır. İşletim sistemi, uygulama odağa sahip olduğunda, belirtilen kod ve verilerin fiziksel bellekte olmasını sağlar. Uygulama odağı kaybettiğinde, kodla veya verilerle ilişkilendirilmiş sayfalar serbest bırakılır. Uygulama odağı yeniden kazandığında, işletim sistemi, uygulama çalışmaya başlamadan önce sayfaları yeniden fiziksel belleğe yükler ve işletim sisteminin, gerektiğinde yumuşak sayfa kilidini geçersiz kılmasına izin verilir.

[31] patentinde bir API için uzaktan kontrol sağlayan yöntemler tanıtılmıştır. Yazılım hareketlerini kontrol eden giriş cihazlarını haritalayan bir API [32] patentinde yer almaktadır. Burada, bilgisayar giriş cihazlarını yazılım uygulamaları ile kullanmak için bir sistem açıklanmıştır. Sistem, giriş aygıtları ile yazılım uygulamaları arasında bir arayüz olarak bir anlambilim dili kullanan bir giriş aygıtı eşleyici API içerir. Giriş aygıtı eşleyicisi, bilgisayara bağlı aygıtların hangi anlambilimlerini uygulayabildiği ve hangi anlambilimin bir kullanıcının yazılımın gerçekleştirmesini isteyebileceği eylemlere karşılık geldiği hakkında bilgi alır. Giriş cihazı eşleyicisi, mevcut cihazlardaki kontrolleri, aynı semantikteki karşılık gelen cihaz kontrolleri ve yazılım eylemlerini mümkün olduğunca yakından eşleştirerek kullanıcının istediği yazılım eylemleriyle eşleştirir. Böylece sistem, giriş cihazlarını ve yazılım uygulamalarını birbirine saydamlaştırır ve eylemlerini kontrol etmek için hangi giriş cihazının kullanıldığına bakılmaksızın çalışacak şekilde tasarlanmış bir yazılım uygulamasına izin verir.

Duyusal olaylar için geliştirilen bir API [33] patentinde yer almaktadır. Buradaki API, bir duyusal alandaki nesnenin koordinat ve hareket bilgilerini sağlar. API bir konum, yer değiştirme, hız, hızlanma ve bir nesnenin duyusal bir alan içinde olduğu sürenin uzunluğunu tanımlamak için yöntemler sağlayabilir. API, en az bir duyusal olay almak için bir olay dinleyicisi ve duyusal olayları işlemek için bir olay işleyicisi içerebilir. Bir GUI (Graphical User Interface – Grafiksel Kullanıcı Arayüzü), temassız gezinme ve kontrol sağlamak için API'yi uygulayabilir.

Literatürdeki çalışmalara bakıldığında, pek çok farklı uygulama için API tasarlanabildiği görülmektedir. Buna göre API'lerin başarılı bir şekilde tasarlanması ve yüksek performans göstermesi, çeşitli uygulamalar için bir kilit nokta haline gelmektedir. Bir API tasarlanırken kullanılacak yöntemler, programlama dilleri vb. kriterler uygun bir şekilde seçildiğinde API'lerin performansının da artacağı görülmektedir.

3. MALZEME VE YÖNTEM

Bu bölümde, çalışmada kullanılan programlama dilleri, uygulanan yöntemler ve kriterler açıklanmaktadır. Bu tez çalışmasında, farklı programlama dillerinde eşdeğer yük testlerine maruz kalacak API uygulamalarının geliştirilerek, performans karşılaştırmalarının yapılması amaçlanmaktadır. Geliştirilen API uygulamalarının performans ölçümleri için uygulanacak yük testleri, k6 ve Locust uygulamaları aracılığıyla yapılmıştır. k6 yük testi uygulaması kaynak kodları EK 1'de, Locust yük testi uygulaması kaynak kodları ise EK 2'de incelenebilir. Testlerin sonuç gözlemi Grafana platformunda gerçekleştirilmiştir. Elde edilen sonuçların çizimi için Grafana platformunun çizim uygulaması ve Microsoft Office Excel 2011 kullanılmıştır.

3.1. UYGULANAN TESTLER

Bu tez çalışmasında geliştirilen API uygulamalarının çeşitli yük altındaki performansları incelenmiştir. Bu amaçla uygulamalara çeşitli yük testleri uygulanmıştır.

Yük testleri, geliştirilen uygulamaları canlı ortama almadan önce, canlı ortama benzer yük ile test etmeye olanak verip, sistemde bulunan darboğazları önceden görmeyi sağlayan araçlardır. Sistemlerin belli şartlar altında limitlerinin tespit edilmesinde önemli rol oynarlar. Bu tez çalışması kapsamında da üç farklı yük testi yapılmıştır. Bu testlerde farklı platformlarda geliştirilen WebAPI uygulamaları yük altında test edilip, sonuçlar birbirleri ile kıyaslanmıştır.

3.1.1. k6 Testi

k6 açık kaynaklı olarak geliştirilen, geliştirici odaklı tasarlanmış, özellikle uygulamaların arkasında çalışan servislerin performanslarını ölçmek için kullanılan bir yük testi uygulamasıdır.

k6 uygulaması, Go dili ile geliştirilmiş olup, test edilecek uygulamalar ile JavaScript betik dosyaları ile haberleşmektedir. Popüler bulut üzerinden yük testi servis sağlayıcısı LoadImpact, arka planda k6 ile ortak çalışmaktadır. k6 temelinde sadelik ve basitlik prensipleriyle geliştirilmiş özellikle InfluxDB gibi açık kaynaklı, zaman serisi tabanlı çalışan veritabanına veri kaydedebilme özelliği ile oldukça iyi bir popülerlik kazanmıştır.

k6 testleri sanal kullanıcı konsepti üzerinde geliştirilip, istenen sayıda sanal kullanıcı ile uygulamalar için yük yaratmaktadırlar. Test edilecek uygulama kullanıcılarının davranışları JavaScript betik dosyaları ile kodlanabilmektedir. Yük testi süresince, test edilecek uygulamaya sanal kullanıcılar, bu davranış dosyaları ile tanımlanan düzende istek gönderirler. k6 test edilmek istenen uygulamayı, istenilen sanal kullanıcı sayısı kadar paralel şekilde çalıştırarak yük yaratıp, test edilen uygulamalarının sanal kullanıcıların yaptığı isteklere verdiği cevapları kaydetmekte, ayrıca bu verileri zaman-serisi temelli veri tabanına kaydedebilmektedir.

k6 testleri faz yapısı kullanılarak belli senaryolar dahilinde planlanabilmektedir. Örneğin bir arka plan servisini test ederken şöyle bir senaryo planlanabilir. Testin ilk 30 saniyesi boyunca sanal kullanıcı sayısı 0'dan 100'e artsın, sonrasında 100 kullanıcı 60 saniye boyunca uygulamaya istekler göndermeye devam etsin, sonrasında kullanıcı sayısı 10 saniye içerisinde 100 sanal kullanıcıdan 0 kullanıcıya düşürülerek test sonlandırılsın. Bu senaryolar yine sanal kullanıcı JavaScript betik dosyaları içerisinde tanımlanabilmektedir. Bu senaryo yapıları kullanılarak uygulamalar canlı ortamlarına benzer şekilde yük testi yapılabilmektedir. Canlı ortamlarda yük dağıtıcılar arkasında çalışan WebAPI uygulamaları genel olarak bu test senaryosuna benzer şekilde ilk önce artan sayıda sonrasında sabit sayıda kullanıcıya hizmet vermektedir.

k6 ayrıca uygulamaların isteklere verdiği cevapları ortalama, medyan, ve farklı yüzdelik dilimlerde raporlayabilmektedir. Böylece uygulama performansı sonuçları incelenirken, çok daha detaylı analizler yapılabilmesine imkan sağlamaktadır.

Bu tez çalışması kapsamında k6 yük testi uygulaması kullanılarak iki farklı test yapılmıştır. Birinci test senaryosunda, uygulamalara 60 saniye boyunca 0'dan 1000'e artacak şekilde yeni sanal kullanıcılar istek yapmışlardır. Bu adımın uygulamaların ısınmalarını, asıl yükü kaldırmalarını sağlamak için hazırlık niteliğindedir. Bu ısınma fazından sonra 1000 sanal kullanıcı uygulamaya 60 saniye boyunca isteklerde bulunmaya devam etmiştir. Bu sırada uygulamaların verdiği cevaplar analiz amacıyla kaydedilmiştir. Son olarak testi sonlandırmak adına 10 saniye boyunca, sanal kullanıcı sayısı 1000'den 0' a düşürülmüştür.

İkinci yük testi uygulamalarının kararlılığını ölçmek için gerçekteştirilmiştir. Bu test kapsamında uygulamalar daha düşük yük altında test edilmiştir. Uygulama yükleri 100 sanal kullanıcı kullanılarak yaratılmış, uygulamalar bu yük altında 60 saniye boyunca test edilip,

sonuçlar kaydedilmiştir. Uygulamalarda anlık oluşabilecek problemleri elimine etmek adına bu testler 5 defa çalıştırılıp sonuçlar ayrı ayrı kaydedilip incelenmiştir.

3.1.2. Locust Testi

Locust kullanımı diğer yük testi uygulamalarına göre daha kolay olarak geliştirilmiş açık kaynaklı bir yük testi uygulamasıdır. Jonatan Heyman, Carl Byström, Joakim Hamrén, Hugo Heyman tarafından web uygulamalarını ve diğer sistemleri test edebilmek amacıyla geliştirilmiştir.

Ürün ismi olan Locust, kelime olarak çekirge anlamını taşımaktadır. Locust testlerinde bu benzetmeden faydalanılmıştır. Yüzlerce, binlerce çekirgenin web uygulamasına aynı anda saldırması benzetilmiştir. Saldırı esnasında Locust, uygulamaların verdiği cevapları kaydederek sistemde potansiyel olarak var olan darboğazların bulunmasını sağlamaktadır. Locust uygulaması olay-tabanlı olarak geliştirilmiştir. Bu yüzden tek bir makine üzerinden binlerce eşzamanlı kullanıcı yükü yaratılabilir.

Locust uygulaması testleri basit Python betik dosyaları ile yapılır. Kullanıcı davranışları Python kodu ile geliştirilerek Locust tarafından yük testine dahil edilir. Olay-tabanlı yapısı sayesinde de binlerce sanal kullanıcı yükü yaratılarak hedef uygulama test edilebilir.

Locust uygulamasının diğer bir özelliği de dağıtık yapıya sahip olması ve ölçeklenebilirliğidir. Tek bir makine üzerinden yaratılan yükün yeterli olmadığı durumlarda birden fazla makine üzerinden test yükü yaratılabilmektedir. Bu durumda testler, ana Locust sunucusunda çalıştırıldıktan sonra, Locust sahip olduğu diğer sunucuları da hesaba katarak, test süresinde birden fazla makinede sanal kullanıcılar yaratarak, hedef uygulamanın test edilmesini sağlar.

Locust uygulaması, testlerin yönetilmesi (testleri listelemek, başlatmak, duraklatmak gibi) ve sonuçların gerçek zamanlı olarak izlenebilmesi için bir web arayüzü desteklemektedir. Bu arayüz kullanılarak test esnasında hedef uygulamanın gösterdiği performans gerçek zamanlı olarak izlenebilmektedir.

Bu tez çalışması kapsamında, Locust uygulaması kullanarak, WebAPI uygulamalarının optimum cevap süresini koruyacak şekilde en fazla kaç sanal kullanıcıya cevap verebildiği test edilmiştir. Bu kapsamda, 250 sanal kullanıcı ile çalıştırılan ilk yük testi sonrasında her bir testte yük, 250 sanal kullanıcı arttırılarak, sırasıyla 500, 750, 1000, 1250, 1500, 1750 ve 2000 sanal

kullanıcı kullanılarak 8 defa çalıştırılmış, uygulamanın verdiği cevap süreleri kaydedilmiştir. Elde edilen rakamlar, farklı ortamlarda geliştirilen WebAPI uygulamalarını kıyaslamak amacıyla analiz edilmiştir.

3.2. PERFORMANS ETMENLERİ

Bir WebAPI uygulamasının performansı, çeşitli etmenlerden dolayı değişiklik gösterebilir. Bu etmenlerin ilk sırasında direkt uygulamaların kendisi ile ilgili etmenler listelenebilir. Örneğin, uygulamada kullanılan ek kütüphaneler, işlemci veya bellek kullanımı gerektiren işlemler, dosya okuma/yazma, ağdan veri çekme gibi birçok operasyon, WebAPI uygulamasının performansını olumlu veya olumsuz yönden etkileyebilir. Bu çalışma kapsamında geliştirilen tüm prototiplerde mümkün olduğu kadar ek bir kütüphane kullanılmamış, uygulamalara gereksiz ek bir özellik eklenmemiş, programlama dillerinin performansını karşılaştırabilmek adına uygulamalar olabilecek en sade halleriyle bırakılmıştır.

Uygulamaların iç yapısı ve tasarımı ile ilgili etmenler dışında, uygulamanın üzerinde çalıştığı uygulama sunucuları da, uygulamaların performansı üzerinde yüksek etkiye sahiptir. Uygulama sunucuları, web uygulamalarının üzerinde çalışması için tasarlanmış sunucu ortamlarıdır. Dünyada en çok bilinen web uygulama sunucuları, IIS (Internet Informaton Service – Internet Bilgi Hizmetleri), Tomcat, Jetty, Glassfish, Weblogic olarak listelenebilir. Özellikle son yıllarda daha sade, gömülü web sunucuları popülarite kazanmaya başlamıştır. Örneğin Node.js uygulamaları daha ilk sürümden itibaren, ek bir uygulama sunucusuna gerek kalmadan gömülü uygulama sunucu ile çalışma özelliğine sahiptir. Sonrasında Java ve .Net platformlarında da benzer destekler geliştirilmeye başlanmıştır. Günümüzde birçok dil ve platform, ek bir uygulama sunucusuna ihtiyaç duymadan, kendi başına çalışabilen web uygulamalarını desteklemekte, bu uygulamalar bir yük dağıtıcının arkasında rahat çalışabilmektedir. Bu çalışma kapsamında da, performansın eş biçimde kıyaslanabilmesi için, WebAPI prototip uygulamaları ek bir uygulama sunucuna gerek kalmayacak şekilde tasarlanmıştır. Tüm prototip uygulamalar kendi başına çalışma özelliğindedir. Bu sayede uygulama sunucularının WebAPI uygulamaları üzerinde yaratabileceği performans etkileri minimum hale getirilmeye çalışılmıştır.

Uygulamaların genel olarak performansını etkileyen diğer bir etmen de üzerlerinde çalıştıkları donanımdır. Bu tez kapsamında yapılan tüm testler, aynı donanım kullanılarak yapılmıştır.

Testler esnasında kullanılan donanıma ek bir cihaz veya aksesuar bağlanmadığı kontrol edilmiştir. Tüm WebAPI uygulamaları, herhangi bir sanallaştırma teknolojisi kullanmadan, gerçek makine üzerinde çalıştırılmıştır. Sadece test sonuçlarının kaydedildiği sistemler için docker konteynerleri kullanılmıştır. Bu sistemler, tüm WebAPI uygulamaları için ortak ve tek olarak kullanılmıştır. WebAPI uygulamaları ile sonuç kaydetme sistemleri arasında herhangi bir ilişki veya bağ yoktur. Bu bağlamda, bunların uygulama performansını etkilememesi sağlanmıştır.

Çalışmada geliştirilen uygulamaların performanslarını karşılaştırabilmek adına uygulamaların yük altında taleplere verdikleri cevap süreleri, bu cevap sürelerinin ortalama ve medyan değerleri, uygulamaların isteklere verebildiği toplam cevap sayısı gibi değerler kaydedilmiştir.

WebAPI uygulamaları için en önemli performans göstergelerinden biri isteklere verilen cevap süresidir. Uygulamalar gelen isteklere farklı yükler altında farklı sürelerde cevap verebilirler. Yapılacak test esnasında da, uygulamalar bazı isteklere yavaş cevap verebilirken, bazı isteklere çok hızlı bir şekilde cevap verebilirler. Eğer testler esnasında sadece ortalama cevap süresi veya medyan cevap süresi incelenirse, bu hızlı ve yavaş verilen cevaplar gözden kaçırılabilir. Bu yüzden çalışma kapsamında, uygulamaların isteklere verdiği ortalama cevap süresi haricinde, yüzdelik dilimlerdeki cevap süreleri de kaydedilmiştir. Yüzdelik dilimlerdeki cevap süreleri p80, p90, p95 şeklinde gösterilmektedir. Örneğin p95'in (%95 yüzdelik dilim) anlamı, toplam gelen tüm isteklerin %95'ine ne kadar sürede cevap verebildiğinin değeridir. Aynı şekilde p99'un anlamı, gelen tüm isteklerin %99'una ne kadar sürede cevap verilebildiğidir.

Örneğin bir uygulama gelen tüm 1000 isteğin %5'ine (50 istek) 300 milisaniyede, geri kalan isteklere de 50 milisaniyede cevap verirse, ortalama cevap süresi 62.5 milisaniye olarak bulunur ve ortalamaya bakıldığında, uygulama performansı yüksek gibi değerlendirilebilir. Fakat aynı sonuçlar için p98 yüzdelik dilimine bakılırsa, bu değer 300 milisaniye olarak görülecektir ve uygulamada yaşanan ani artış ve düşüşler bu sayede gözden kaçmadan incelenebilecektir.

3.3. PROGRAMLAMA DİLLERİ

Bu tez çalışmasında, C#, Java, Go, Python ve Node.js programlama dilleri kullanılarak çeşitli uygulamalar gerçekleştirilmiş ve bu uygulamaların performansları ile ilgili çeşitli karşılaştırmalar yapılmıştır. Bu programlama dillerini belli başlı özellikleri aşağıda verilmiştir:

C# programlama dili, Microsoft tarafından Anders Hejlsberg liderliğinde geliştirilmiştir. C ve C++ kod sözdizimine benzer bir kod yapısındadır. C# dili .Net çatı platformu için hazırlanmış tamamen nesne yönelimli bir dilidir. C# dili ile geliştirilen uygulamalar derlendiğinde ara bir dile (Intermadiate Language – IL), çevrilmektedir. Bu ara dile Microsoft Ara Dili (Microsoft Intermadiate Language – MSIL) denmektedir. Uygulamanın çalıştırılması Ortak Çalışma Zamanı (Common Language Runtime – CLR) tarafından yapılır. CLR oluşturulan MSIL dosyasını okuyup yorumlayarak makine diline çevirmekte ve uygulamayı çalıştırmaktadır. C# ve .Net platformu üzerinde Windows uygulamaları, web uygulamaları, Windows Azure ile bulut uygulamaları, Microsoft Office için eklentiler, veri tabanı uygulamaları geliştirilebilmektedir.

Java programlama dili, Sun Microsystems mühendislerinden James Gosling tarafından 1995 yılında geliştirilmeye başlanmıştır. Java programlama dili, C ve C++'dan birçok sözdizimini almaktadır. Buna rağmen Java, daha basit nesne modeli ile daha az düşük seviye olanaklar içerir. Java programlama dili ile birçok farklı türde uygulama geliştirilebilmektedir. Dağıtık sistemler, web uygulamaları, web servisleri, web uygulama programlama arayüzleri ve mobil uygulamalar Java programlama dili ile geliştirilebilmektedir. Java dilinin en önemli özelliklerinden biri taşınabilir olması, platform bağımsız olmasıdır. Java uygulamaları bilgisayar mimarisinden bağımsız olarak Java Sanal Makinesi (Java Virtual Machine – JVM) içerisinde çalışmaktadır. Bu bağlamda, aynı uygulama Windows işletim sistemi üzerinde çalışabiliyorsa, tekrar derlemeye gerek kalmadan Linux işletim sistemi üzerinde de çalışabilmektedir. Bunun gerçekleşmesini sağlayan da JVM'dir. Java dili ile geliştirilen kod derlendiğinde sanal makine koduna (byte code) dönüştürülmektedir. Bu sanal makine kodu, sanal makine tarafından adım adım işletilerek çalıştırılır. Bu yüzden de sanal makinenin çalıştığı tüm platformlarda Java uygulamaları çalışabilmektedir.

Go programlama dili, 2007 yılında Google tarafından, Ken Thompson, Rob Pike, Robert Griesiemer önderliğinde geliştirilme başlanmıştır. Golang ismiyle de tasvir edilmektedir. Kararlı ilk sürümü, Go 1.0, 28 Mart 2012 yılında yayınlanmıştır. Go açık kaynaklı ve tamamen ücretsiz bir programlama dilidir. Bunun dışında, Go, derlenmiş, statik veri tipleri kullanan bir dildir. Go dilinin birincil kullanım yeri sistem programlamadır. Bunun yanında getirdiği yüksek performans ve eş zamanlı programlama özellikleriyle yüksek performanslı web uygulamaları

geliştirmek amacıyla da kullanılabilmektedir. Go dili söz dizilimini öğrenmek basittir, sadece 25 adet komut içermektedir.

Python dilinin geliştirilmesine 1990 yılında Guido van Rossum tarafından Amsterdam'da başlanmıştır. Python genel amaçlı programlama için kullanılan yüksek seviye bir programlama dilidir. Python, otomatik bellek yönetimi özelliği bulunan dinamik tipte bir sistem barındırır, nesne yönelimli, fonksiyonel ve yordamsal gibi birden çok programlama paradigmasını destekler. Geniş ve kapsamlı bir standart kütüphanesi bulunmaktadır. Python dili ile geliştirilen uygulamaların derlenmesine gerek yoktur. Python uygulamaları çalıştırıldığı zaman kaynak kodlar bir yorumlayıcı tarafından işleme alınır. Bu durum, PERL ve PHP'de kullanılan web yorumlayıcılarına benzer. Python dili hemen hemen her türlü platformda çalışabilir. (Unix, Linux, Mac, Windows). Girintilere dayalı basit sözdizimi, dilin öğrenilmesini ve akılda kalmasını kolaylaştırır. Bu da ona söz diziliminin ayrıntıları ile vakit yitirmeden programlama yapılmaya başlanabilen bir dil olma özelliği kazandırır. Python ile sistem programlama, kullanıcı arabirimi programlama, ağ programlama, uygulama ve veritabanı yazılımı programlama gibi birçok alanda yazılım geliştirilebilir. Özellikle 2000'li yıllardan itibaren bilimsel veya mühendislikle ilgili hesaplamalı çalışmalarda da çokça kullanılmaya başlamıştır. Bunda hem donanımsal hem de yazılımsal gelişmelerin etkisi olmuştur. Python internet ve onunla beraber açık kaynak kod veya özgür yazılım akımının yaygınlık kazanması, bu yaklaşımı benimseyen bilimcilerin internet üzerinden ve geliştirilen verimli araçlar sayesinde, günümüzde oldukça büyük bir popülerite kazanmıştır.

Node.js Ryan Dahl tarafından 2009 yılında geliştirilmeye başlanan bir programlama dilidir. Node.js, sunucu tarafında JavaScript kodu çalıştıran, açık kaynaklı bir platformlar arası JavaScript çalışma ortamıdır. Node.js programlama dili, yazılım geliştiricilerin JavaScript'i sunucu tarafındaki yazımda dinamik web sayfası içeriğini, sayfa kullanıcının web tarayıcısına iletilmeden önce oluşturmasını sağlar. Bu sayede hem sunucu hem de istemci tarafında farklı programlama dillerinin kullanılmasının önüne geçilmiş olur. Node.js kullanarak genel olarak yüksek performanslı web uygulamaları ve bilgisayar ağı uygulamaları geliştirilebilir. Node.js programlama arayüzleri, dosya işlemleri, bilgisayar ağı işlemleri, kriptografi, veri akışları ve diğer temel fonksiyonalitelerini kolaylaştıracak şekilde geliştirilmiştir. Node.js'in gerçekleştirbildiği bilgisayar ağı uygulamaları arasında DNS (Domain Name System – Alan Adı Sistemi), TCP (Transmission Control Protocol – İletişim Kontrol Protokolü), TLS

(Transport Layer Security – Taşıma Katmanı Güvenliği), SSL (Secure Sockets Layer – Güvenli Soket Katmanı), UDP (User Datagram Protocol – Kullanıcı Datagram Protokolü) sayılabilir. Node.js Linux, macOS, Microsoft Windows, SmartOS, FreeBSD, ve IBM AIX platformlarda çalışabilmektedir. Node.js olay yönelimli programlama konseptini web uygulamalarına getirmiştir.

4. BULGULAR

Bu bölümde performans kıyaslaması için geliştirilen uygulamalar açıklanmaktadır. Çalışmada ulaşılan sonuçların elde edilmesinde .NET Core ortamıyla C#, Java, Go, Python ve Node.js dilleri kullanılmıştır. Her bir programlama dili için 2 adet k6 test seti, bir adet Locust test seti olmak üzere toplamda 3 adet test seti uygulanmıştır. Yapılan ilk yük testinde k6 adlı uygulama baz alınmıştır. Sanal kullanıcı altyapısı sağlayan bu uygulamada, bu sanal kullanıcıların davranışları kodlanarak uygulamalar yük altında test edilebilmektedir. Geliştirilen WebAPI uygulamaları için sanal kullanıcı kodlamaları EK 1'de verilmiştir. Bu kapsamda, 100 milisaniye bekleyip WebAPI uygulamasına yeni bir istekte bulunan kullanıcı davranışı kodlanmıştır. k6 uygulaması ile iki farklı test gerçekleştirilmiştir.

İlk test kapsamında, WebAPI uygulamalarının önbelleklemelerini sağlıklı yapabilmeleri için 60 saniye boyunca sanal kullanıcı sayısı 0'dan 1000'e kadar arttırılmış, sayının 1000'e ulaşmasının ardından, uygulama 60 saniye boyunca bu yük altında çalıştırılmıştır. Sonrasında da testi sonlandırmak adına, 10 saniye boyunca kullanıcı sayısı 1000'den 0'a düşmesi beklenmiştir. Toplamda 130 saniye süren test boyunca, uygulamaların isteklere verdiği cevapların performansı ölçülmüştür. Bu test, geliştirilen 5 farklı WebAPI uygulaması üzerinde uygulanmış, sonuçları kayıt altına alınmıştır.

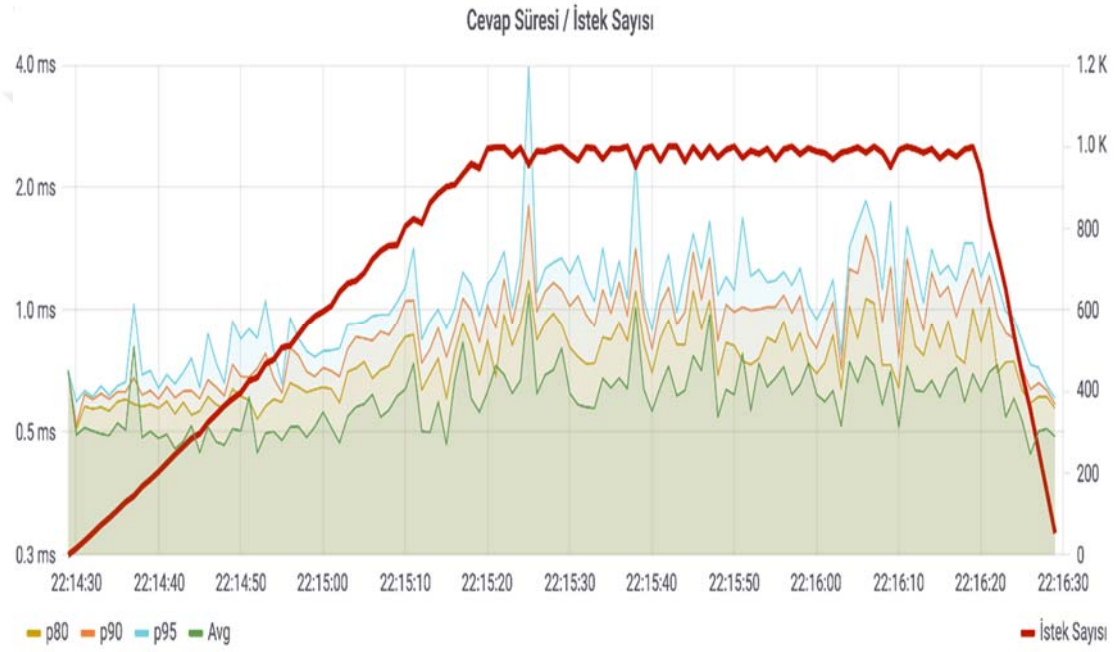
k6 uygulaması kullanılarak gerçekleştirilen ikinci testte ise k6 uygulamasının 100 sanal kullanıcıya kadar çıkması beklenip, 100 sanal kullanıcıya ulaşıldığı nokta sonrasında test 60 saniye boyunca koşularak, uygulamalarının yapılan isteklere verdiği cevap süreleri ölçülmüştür. Bu test her uygulama için test 5 defa çalıştırılmış ve sonuçları kayıt altına alınmıştır.

İkinci test setinde Locust adlı uygulama baz alınmıştır. Locust uygulamasının sağladığı sanal kullanıcı kavramından faydalanarak, geliştirilen uygulamalara 60 saniye boyunca talep gönderilmiş, uygulamanın taleplere verdiği başarılı cevaplar ve cevap süreleri kayıt altına alınmıştır. Bu test her prototip WebAPI uygulaması için 8 defa tekrar edilmiştir. İlk teste 250 sanal kullanıcı ile başlanmış ve her seferinde 250 yeni sanal kullanıcı eklenmiştir, buna göre test, 250-2000 arasında değişen bir sanal kullanıcı sayısı aralığı için çalıştırılmıştır. Locust Yük Test Uygulaması'nın kodları Ek 2'de sunulmuştur.

4.1. UYGULAMALAR

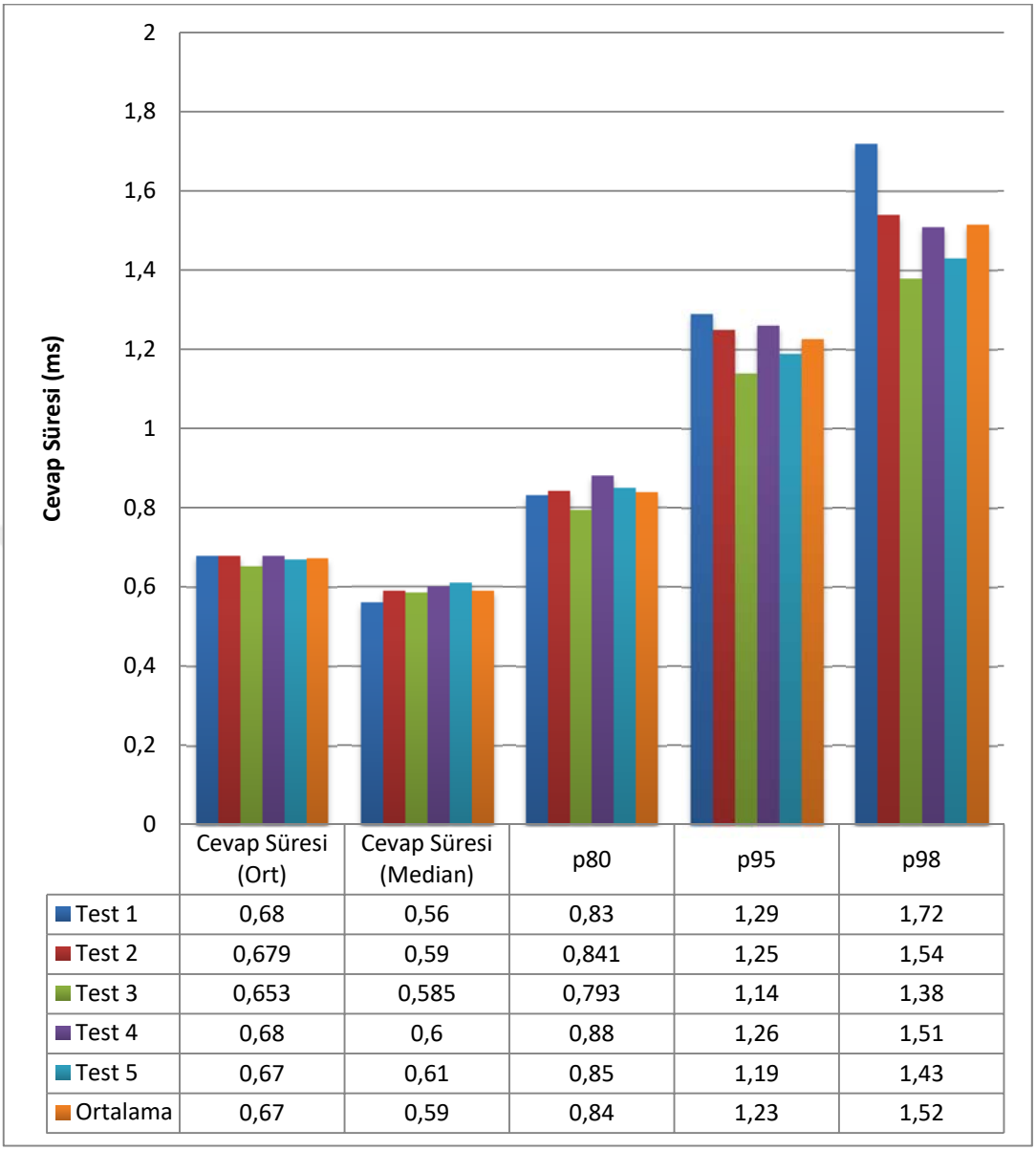
4.1.1. C# Programlama Dili ile Rest API Uygulaması

Tez çalışmasında ilk olarak C# programlama dili kullanılarak Rest API uygulaması geliştirilmiştir. Bu programlama dilinde tasarlanan uygulamanın kaynak kodları EK 3'te incelenebilir. Şekil 4.1'de, k6 yük testi uygulamasının sonuçları gözlemlenebilir. Doğrusal olarak istemci sayısı artışı uygulanarak, 1000 birimde sabitlenmiş ve cevap süresi sonuçları kaydedilmiştir.



Şekil 4.1: C# programlama dili cevap süresi grafiği (k6 yük testi)

Yapılan bu testte, kullanıcı sayısı 0'dan 1000'e doğrusal olarak arttırılırken, ortalama cevap süresi 0.5ms ile 0.8ms arasında gözlemlenmiştir. Test, kullanıcı sayısı 1000'e ulaştıktan sonra 1 dakika boyunca devam ettirildiğinde ise, ortalama cevap süresinin 0.5ms ile 1ms aralığında olduğu gözlemlenmiştir. Servisin performansını daha detaylı olarak gösteren p95 cevap süresi metriği incelendiğinde ise, anlık olarak 4ms'e çıkmış olmasına rağmen, test boyunca 0.75ms – 2ms aralığında seyrettiği ve bu bağlamda uygulamanın performansını sürdürülebilir şekilde koruyabildiği gözlemlenmiştir. Şekil 4.2'de k6 yük testi için cevap süresi grafiği bulunmaktadır. Aynı şartlarda 5 kere tekrarlanarak, ortalama cevap süresi, medyan cevap süresi, minimum cevap süresi, maksimum cevap süresi, p80 orantılı cevap süresi, p95 orantılı cevap süresi ve p98 orantılı cevap süresi kaydedilmiştir.



	Cevap Süresi (Ort)	Cevap Süresi (Median)	p80	p95	p98
Test 1	0,68	0,56	0,83	1,29	1,72
Test 2	0,679	0,59	0,841	1,25	1,54
Test 3	0,653	0,585	0,793	1,14	1,38
Test 4	0,68	0,6	0,88	1,26	1,51
Test 5	0,67	0,61	0,85	1,19	1,43
Ortalama	0,67	0,59	0,84	1,23	1,52

Şekil 4.2: C# programlama dili cevap süresi grafiği (k6 yük testi)

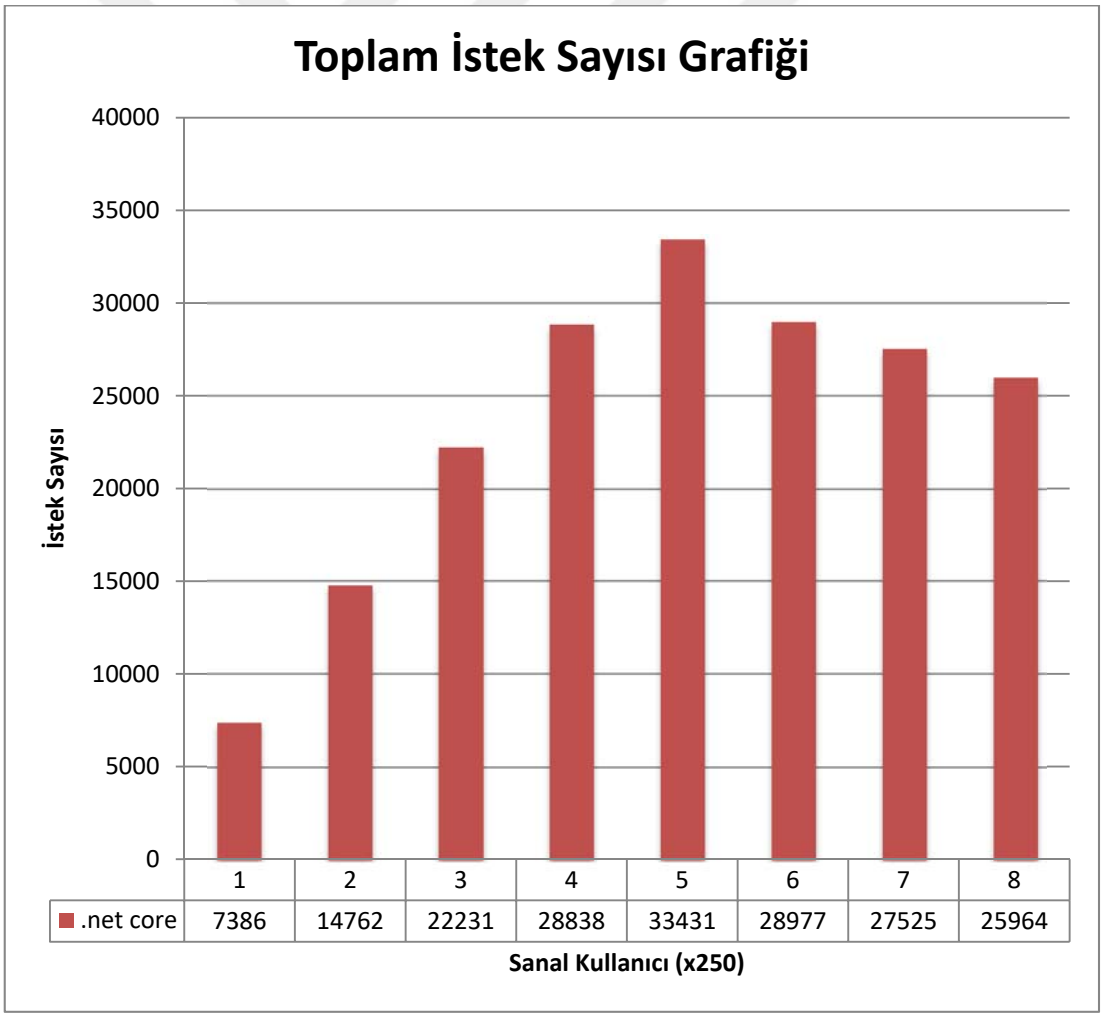
Bu grafiğe bakıldığında,

- tüm testler için ortalama cevap sürelerinin birbirine oldukça yakın olduğu ve 0,65ms - 0,68ms aralığında gerçekleştiği,
- tüm testler için medyan cevap sürelerinin birbirine oldukça yakın olduğu ve 0,56ms - 0,61ms aralığında gerçekleştiği,
- tüm testler için p80 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 0,79ms - 0,85ms aralığında gerçekleştiği,

- tüm testler için p95 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 1,14ms - 1,29ms aralığında gerçekleştiği,
- tüm testler için p98 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 1,38ms - 1,72ms aralığında gerçekleştiği,
- en düşük ve en yüksek cevap sürelerinin farklı kriterlere göre (ortalama cevap süresi, medyan cevap süresi, p80 orantılı cevap süresi, p95 orantılı cevap süresi, p98 orantılı cevap süresi) farklı testlerde ortaya çıktığı,

görülmektedir.

Şekil 4.3'te Locust yük testi ile uygulamanın cevaplayabildiği toplam istek sayısının, istek yapan istemci sayısına göre değişimi kaydedilmiştir.



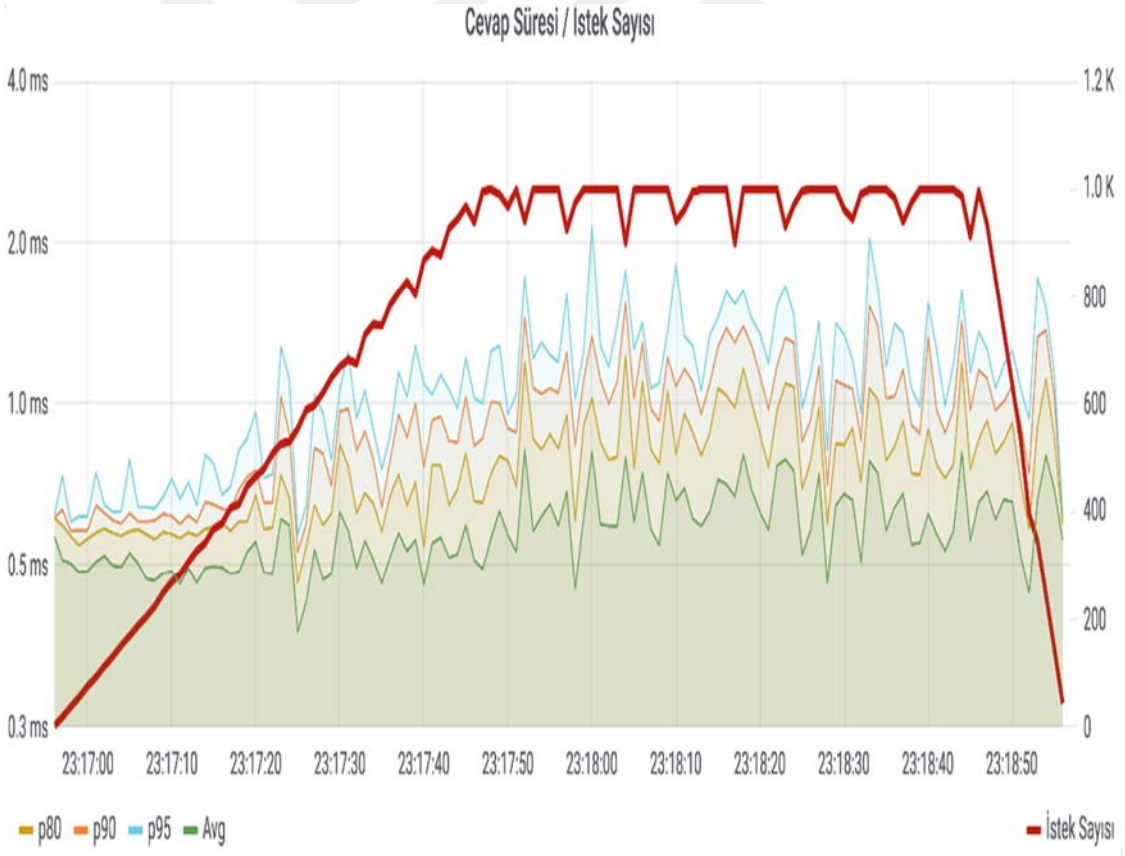
Şekil 4.3: C# programlama dili istek sayısı grafiği (Locust yük testi)

Bu grafiğe bakıldığında, C# uygulamasının, 1250 istemciye kadar verebildiği başarılı istek sayısını arttırdığı (33431 istek), 1250 istemci sonrası ise, cevaplayabildiği istek sayısının artmadığı ve hatta azaldığı gözlemlenmiştir. Uygulamanın, kullanılan donanım ile optimum noktanın 1250 kullanıcı ile yapılan testte, toplam 33431 isteğe cevap vererek sağladığı gözlemlenmiştir.

4.1.2. Java Programlama Dili ile Rest API Uygulaması

Günümüzde pek çok alanda kullanılmakta olan Java programlama dili, bu tez çalışmasında Rest API uygulaması geliştirilirken kullanılan ikinci programlama dilidir. Bu programlama dilinde tasarlanan uygulamanın kaynak kodları EK 4'te incelenebilir.

Şekil 4.4'te, k6 yük testi uygulamasının sonuçları gözlemlenebilir. Doğrusal olarak istemci sayısı artışı uygulanarak, 1000 birimde sabitlenmiş ve cevap süresi sonuçları kaydedilmiştir.

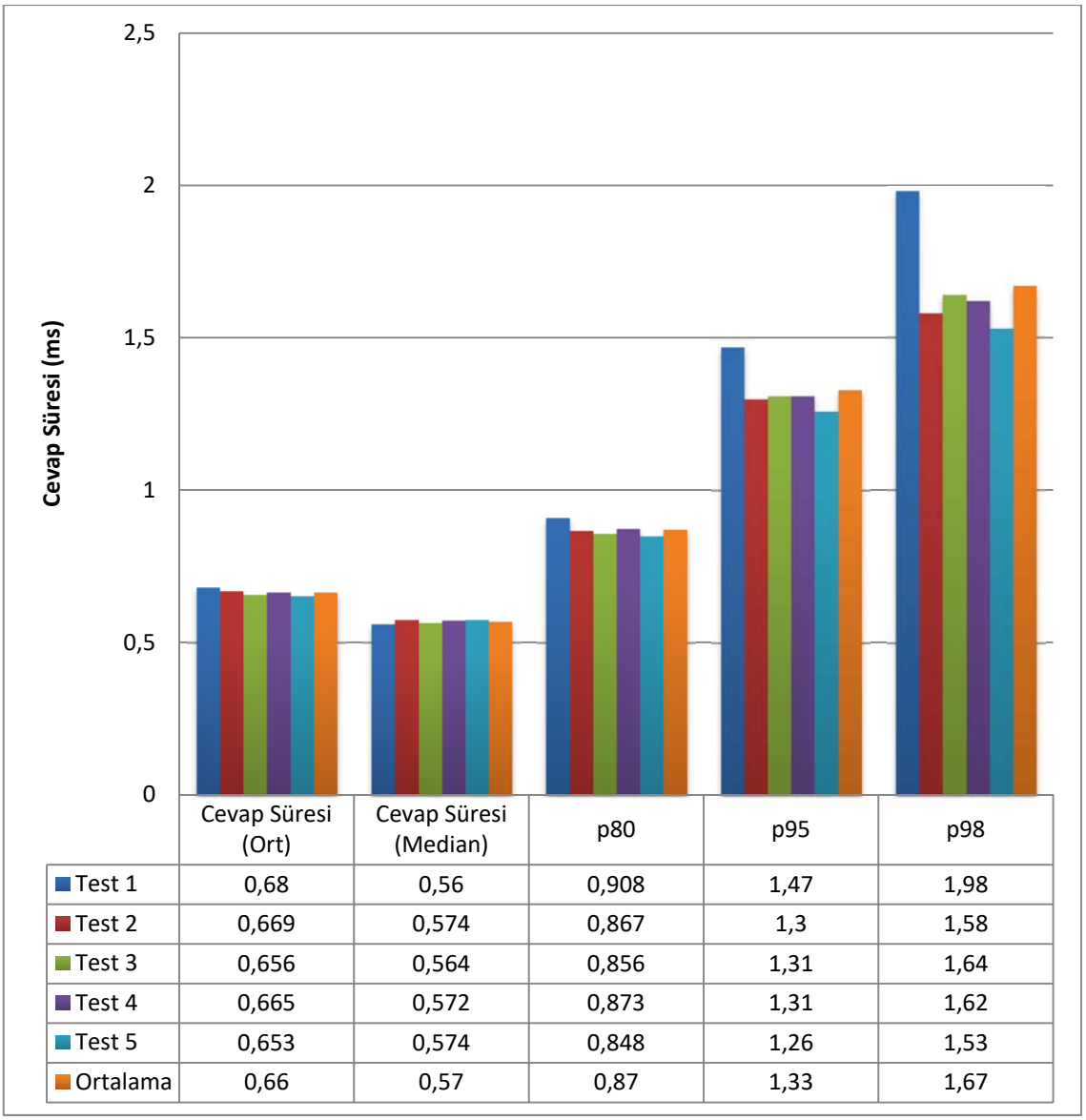


Şekil 4.4: Java programlama dili cevap süresi grafiği (k6 yük testi)

Yapılan testte, kullanıcı sayısı 0'dan 1000'e doğrusal olarak arttırıldığı sürede, ortalama cevap süresinin 0.35ms ile 0.6ms arasında olduğu; istemci sayısı 1000'e ulaştıktan sonra

1 dakika boyunca devam ettirildiği sürede ise, ortalama cevap süresinin 0.45ms ile 0.8ms aralığında gerçekleştiği gözlemlenmiştir. Servisin performansını daha detaylı olarak gösteren p95 cevap süresi metriği incelendiğinde ise, anlık olarak en fazla 2ms'e çıktığı, rağmen, test boyunca 0.75ms – 1.5ms aralığında seyrettiği gözlemlenmiştir.

Şekil 4.5'te k6 yük testi için cevap süresi grafiği bulunmaktadır. Aynı şartlarda 5 kere tekrarlanarak, ortalama cevap süresi, medyan cevap süresi, minimum cevap süresi, maksimum cevap süresi, p80 orantılı cevap süresi, p95 orantılı cevap süresi ve p98 orantılı cevap süresi kaydedilmiştir.



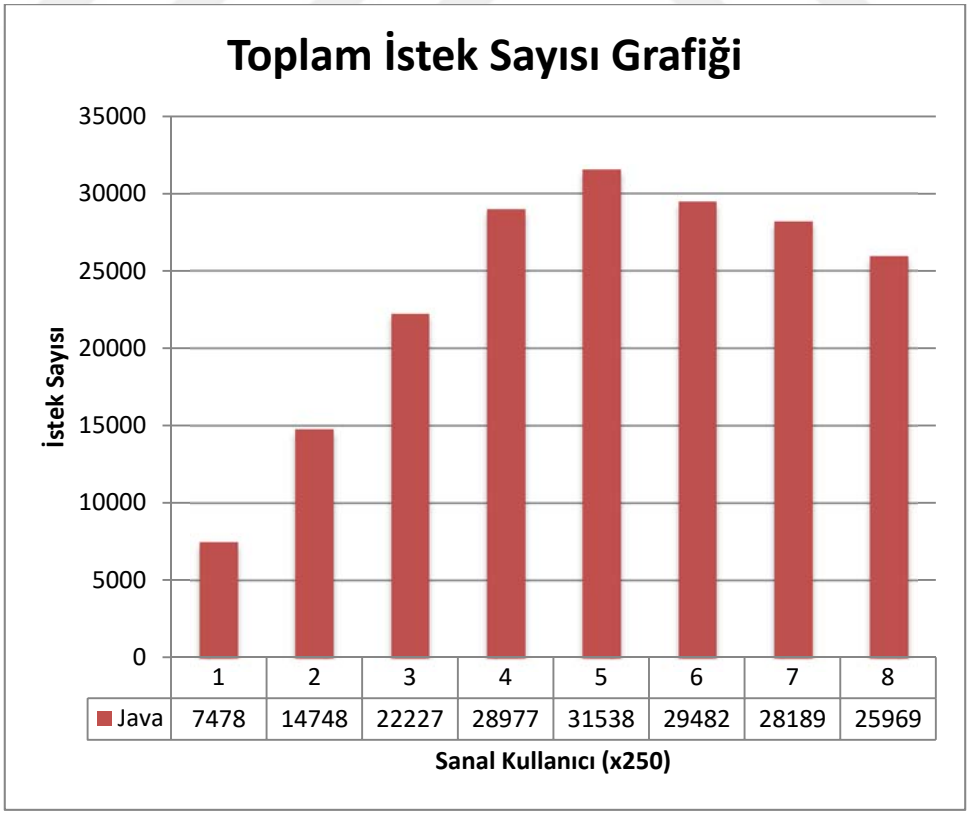
	Cevap Süresi (Ort)	Cevap Süresi (Median)	p80	p95	p98
Test 1	0,68	0,56	0,908	1,47	1,98
Test 2	0,669	0,574	0,867	1,3	1,58
Test 3	0,656	0,564	0,856	1,31	1,64
Test 4	0,665	0,572	0,873	1,31	1,62
Test 5	0,653	0,574	0,848	1,26	1,53
Ortalama	0,66	0,57	0,87	1,33	1,67

Şekil 4.5: Java programlama dili cevap süresi grafiği (k6 yük testi)

Bu grafiğe bakıldığında,

- tüm testler için ortalama cevap sürelerinin birbirine oldukça yakın olduğu ve 0,66ms - 0,68ms aralığında gerçekleştiği,
- tüm testler için medyan cevap sürelerinin birbirine oldukça yakın olduğu ve 0,56ms - 0,57ms aralığında gerçekleştiği,
- tüm testler için p80 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 0,84ms - 0,91ms aralığında gerçekleştiği,
- tüm testler için p95 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 1,26ms - 1,47ms aralığında gerçekleştiği,
- tüm testler için p98 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 1,53ms - 1,98ms aralığında gerçekleştiği,
- en düşük cevap sürelerinin genellikle Test 5'te ortaya çıktığı,

görülmektedir. Şekil 4.6'da Locust yük testi ile uygulamanın cevaplayabildiği toplam istek sayısının, istek yapan istemci sayısına göre değişimi kaydedilmiştir.



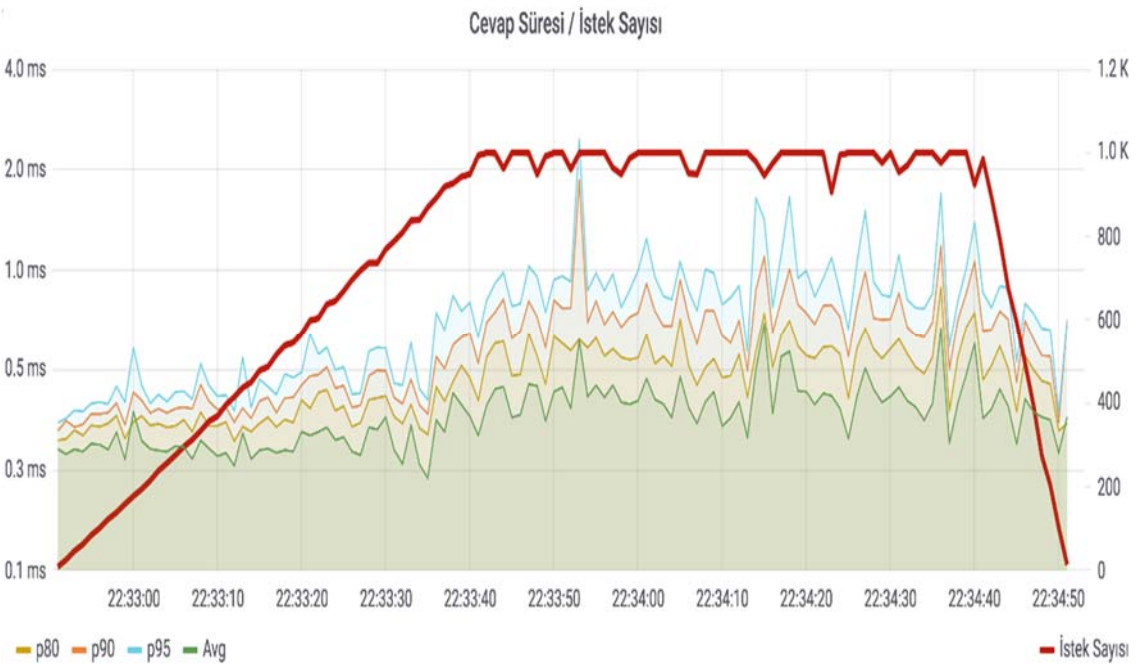
Şekil 4.6: Java programlama dili istek sayısı grafiği (Locust yük testi)

Testte, Java uygulamasının, 1250 istemciye kadar verebildiği başarılı istek sayısını arttırdığı, 1250 istemci sonrası ise, cevap sayısının artmadığı ve hatta azaldığı gözlemlenmiştir. Uygulamanın, kullanılan donanım ile optimum noktanın 1250 kullanıcı ile yapılan testte, toplam 31538 isteğe cevap vererek sağladığı gözlemlenmiştir. Sonraki testlerde uygulamanın başarılı cevap verdiği istek sayısı azalmıştır.

4.1.3. Go Programlama Dili ile Rest API Uygulaması

Go programlama dili ile geliştirilen Rest API uygulamasının kaynak kodları EK 5'te incelenebilir.

Şekil 4.7'de, k6 yük testi uygulamasının sonuçları gözlemlenebilir. Doğrusal olarak istek sayısı artışı uygulanarak, 1000 birimde sabitlenmiş ve cevap süresi sonuçları kaydedilmiştir.

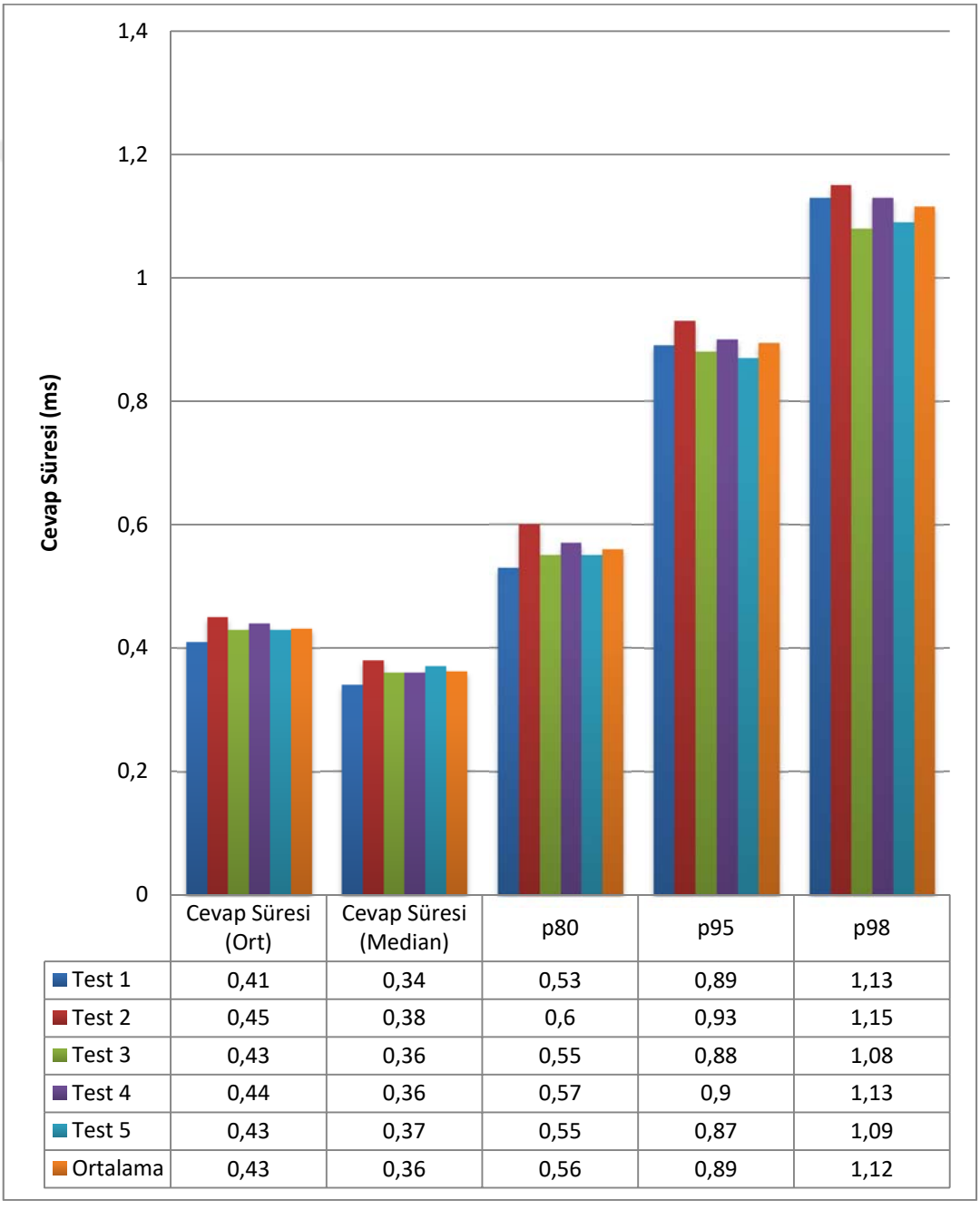


Şekil 4.7: Go programlama dili cevap süresi grafiği (k6 yük testi)

Yapılan bu testte, kullanıcı sayısı 0'dan 1000'e doğrusal olarak arttırılırken, ortalama cevap süresi 0.3ms ile 0.45ms arasında olduğu, kullanıcı sayısı 1000'e ulaştıktan sonra 1 dakika boyunca devam ettirildiğinde ise, ortalama cevap süresinin 0.3ms ile 0.7ms aralığında gerçekleştiği gözlemlenmiştir. Servisin performansını daha detaylı olarak gösteren p95 cevap süresi metriği incelendiğinde ise, anlık olarak 2ms'e çıkmış olmasına rağmen, test boyunca

0.5ms – 1.5ms aralığında seyrettiği gözlemlenmiştir. Bu bağlamda uygulamanın performansını sürdürülebilir şekilde koruyabildiği söylenebilir.

Şekil 4.8'de k6 yük testi için cevap süresi grafiği bulunmaktadır. Aynı şartlarda 5 kere tekrarlanarak, ortalama cevap süresi, medyan cevap süresi, minimum cevap süresi, maksimum cevap süresi, p80 orantılı cevap süresi, p95 orantılı cevap süresi ve p98 orantılı cevap süresi kaydedilmiştir.



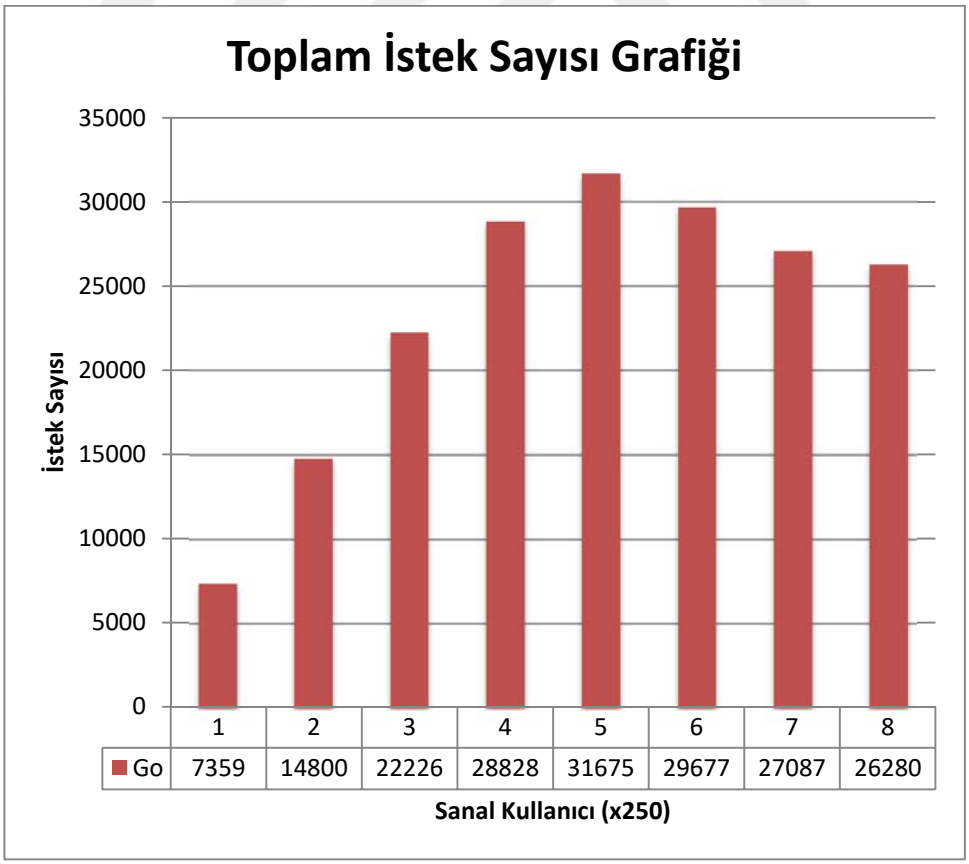
	Cevap Süresi (Ort)	Cevap Süresi (Median)	p80	p95	p98
Test 1	0,41	0,34	0,53	0,89	1,13
Test 2	0,45	0,38	0,6	0,93	1,15
Test 3	0,43	0,36	0,55	0,88	1,08
Test 4	0,44	0,36	0,57	0,9	1,13
Test 5	0,43	0,37	0,55	0,87	1,09
Ortalama	0,43	0,36	0,56	0,89	1,12

Şekil 4.8: Go programlama dili cevap süresi grafiği (k6 yük testi)

Bu grafiğe bakıldığında,

- tüm testler için ortalama cevap sürelerinin birbirine oldukça yakın olduğu ve 0,41ms - 0,45ms aralığında gerçekleştiği,
- tüm testler için medyan cevap sürelerinin birbirine oldukça yakın olduğu ve 0,34ms - 0,38ms aralığında gerçekleştiği,
- tüm testler için p80 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 0,53ms - 0,60ms aralığında gerçekleştiği,
- tüm testler için p95 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 0,87ms - 0,93ms aralığında gerçekleştiği,
- tüm testler için p98 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 1,09ms - 1,15ms aralığında gerçekleştiği,

görülmektedir. Şekil 4.9'da Locust yük testi ile uygulamanın cevaplayabildiği toplam istek sayısının, istek yapan istemci sayısına göre değişimi kaydedilmiştir.



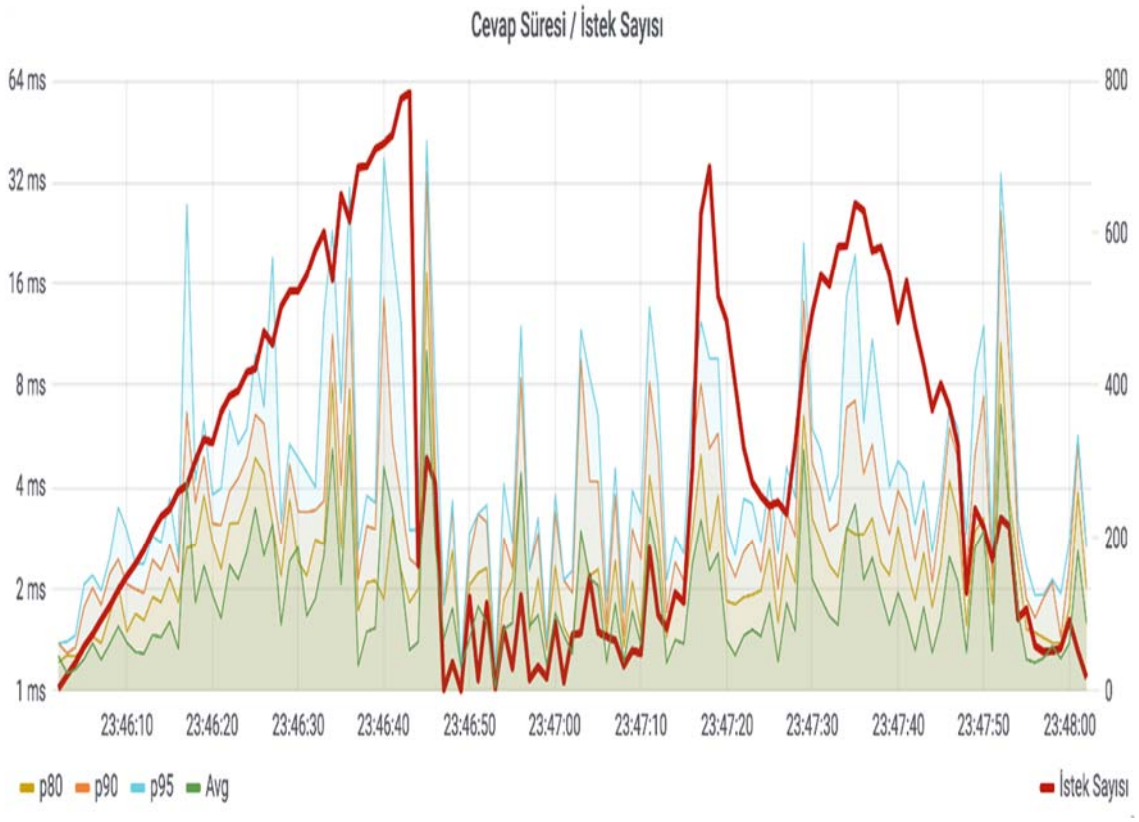
Şekil 4.9: Go programlama dili istek sayısı grafiği (Locust yük testi)

Testte, Go uygulamasının, 1250 istemciye kadar verebildiği başarılı istek sayısını arttırdığı, 1250 istemci sonrası ise, cevap sayısının artmadığı ve hatta azaldığı; uygulamanın, kullanılan donanım ile optimum noktanın 1250 kullanıcı ile yapılan testte, toplam 31675 isteğe cevap vererek sağladığı; 1500, 1750 ve 2000 sanal kullanıcı ile yapılan testlerde, uygulamanın daha az sayıda isteğe cevap verebildiği gözlemlenmiştir.

4.1.4. Python Programlama Dili ile Rest API Uygulaması

Tez çalışmasında Rest API uygulamalarının geliştirildiği bir diğer programlama dili, günümüzde gittikçe popülerleşen Python dilidir. Bu programlama dilinde geliştirilen uygulamanın kaynak kodları EK 6'da incelenebilir.

Şekil 4.10'da, k6 yük testi uygulamasının sonuçları gözlemlenebilir.

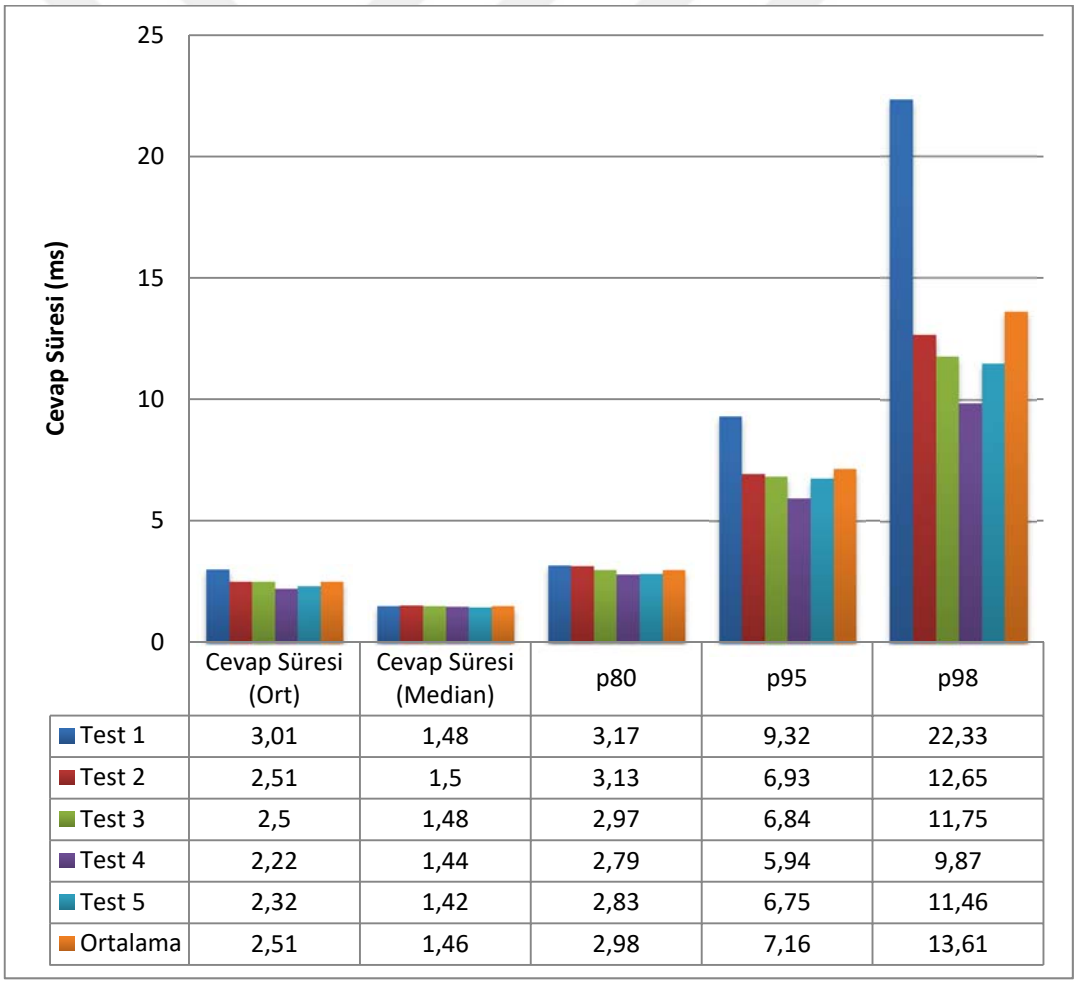


Şekil 4.10: Python programlama dili cevap süresi grafiği (k6 yük testi)

Yapılan testte, kullanıcı sayısı 0'dan 1000'e doğrusal olarak arttırılırken, ortalama cevap süresi 0.5 ms ile 8 ms arasında olduğu, kullanıcı sayısı 1000'e ulaştıktan sonra 1 dakika boyunca devam ettirildiğinde ise, ortalama cevap süresinin 0.1ms ile 8ms aralığında dalgalandığı

gözlemlenmiştir. Servisin performansını daha detaylı olarak gösteren p95 cevap süresi metriğini incelediğimizde ise, anlık olarak 32ms'e kadar çıktığı, test süresince de 2ms – 40ms aralığında seyrettiği görülmüştür. Bu bağlamda uygulamanın performansını sürdürülebilir şekilde koruyamadığı, cevap sürelerinin ve cevap verilen istek sayısının test süresince ciddi bir şekilde dalgalandığı gözlemlenmiştir. Bu bağlamda Python ile geliştirilen uygulamanın diğer dillere göre daha kötü performans gösterdiği söylenebilir.

Şekil 4.11'de k6 yük testi için cevap süresi grafiği bulunmaktadır. Aynı şartlarda 5 kere tekrarlanarak, ortalama cevap süresi, medyan cevap süresi, minimum cevap süresi, maksimum cevap süresi, p80 orantılı cevap süresi, p95 orantılı cevap süresi ve p98 orantılı cevap süresi kaydedilmiştir.



	Cevap Süresi (Ort)	Cevap Süresi (Median)	p80	p95	p98
Test 1	3,01	1,48	3,17	9,32	22,33
Test 2	2,51	1,5	3,13	6,93	12,65
Test 3	2,5	1,48	2,97	6,84	11,75
Test 4	2,22	1,44	2,79	5,94	9,87
Test 5	2,32	1,42	2,83	6,75	11,46
Ortalama	2,51	1,46	2,98	7,16	13,61

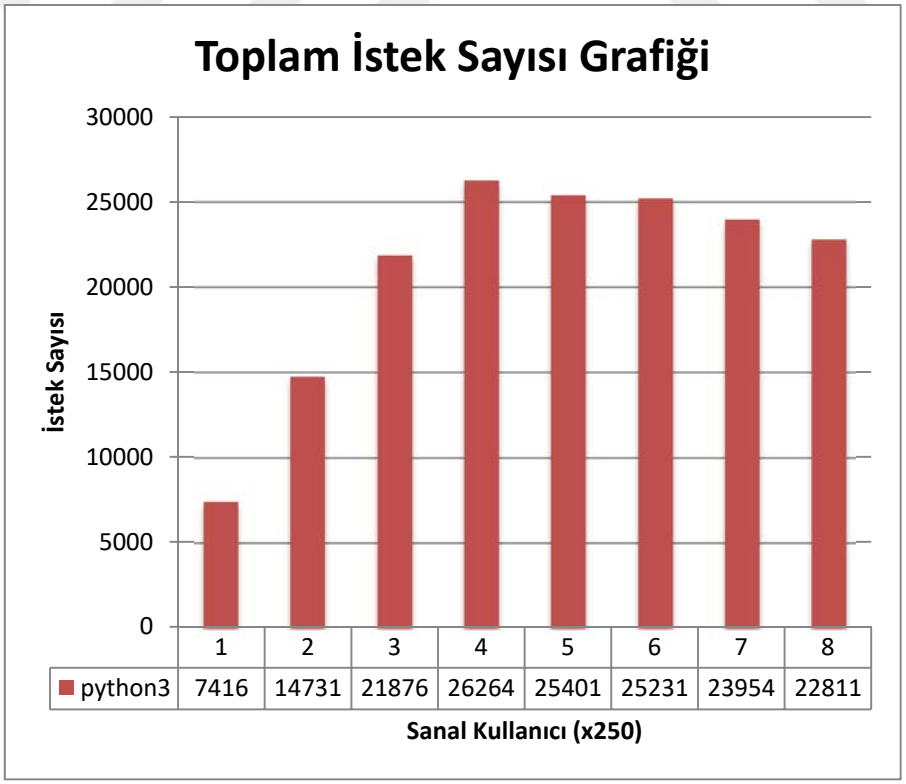
Şekil 4.11: Python programlama dili cevap süresi grafiği (k6 Yük Testi)

Bu grafiğe bakıldığında,

- tüm testler için ortalama cevap sürelerinin birbirine oldukça yakın olduğu ve 2,22ms - 3,01ms aralığında gerçekleştiği,
- tüm testler için medyan cevap sürelerinin birbirine oldukça yakın olduğu ve 1,42ms - 1,5ms aralığında gerçekleştiği,
- tüm testler için p80 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 2,79ms - 3,17ms aralığında gerçekleştiği,
- tüm testler için p95 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 5,94ms - 9,32ms aralığında gerçekleştiği,
- tüm testler için p98 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 9,87ms - 22,33ms aralığında gerçekleştiği,

görülmektedir.

Şekil 4.12'de Locust yük testi ile uygulamanın cevaplayabildiği toplam istek sayısının, istek yapan istemci sayısına göre değişimi kaydedilmiştir.



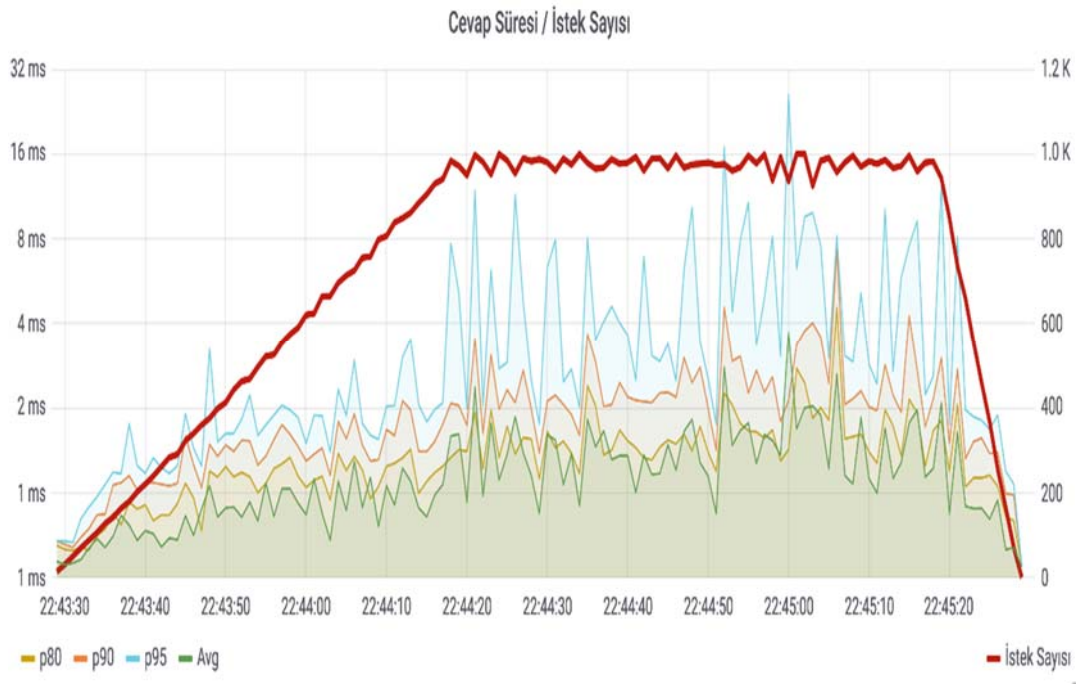
Şekil 4.12: Go programlama dili istek sayısı grafiği (Locust yük testi)

Bu grafiğe bakıldığında, Python uygulamasının, 1000 istemciye kadar verebildiği başarılı istek sayısını arttırdığı, 1000 istemci sonrası ise, cevap sayısının artmadığı ve hatta azaldığı gözlemlenmiştir. Uygulamanın, kullanılan donanım ile optimum noktanın 1000 kullanıcı ile yapılan testte, toplam 26264 isteğe cevap vererek sağladığı gözlemlenmiştir. Örneğin 2000 kullanıcı ile yapılan testte uygulama sadece 22811 isteğe cevap verebilmiştir.

4.1.5. Node.js Programlama Dili ile Rest API Uygulaması

Node.js dilinde geliştirilen WebAPI uygulamasının kaynak kodları EK 7'de incelenebilir.

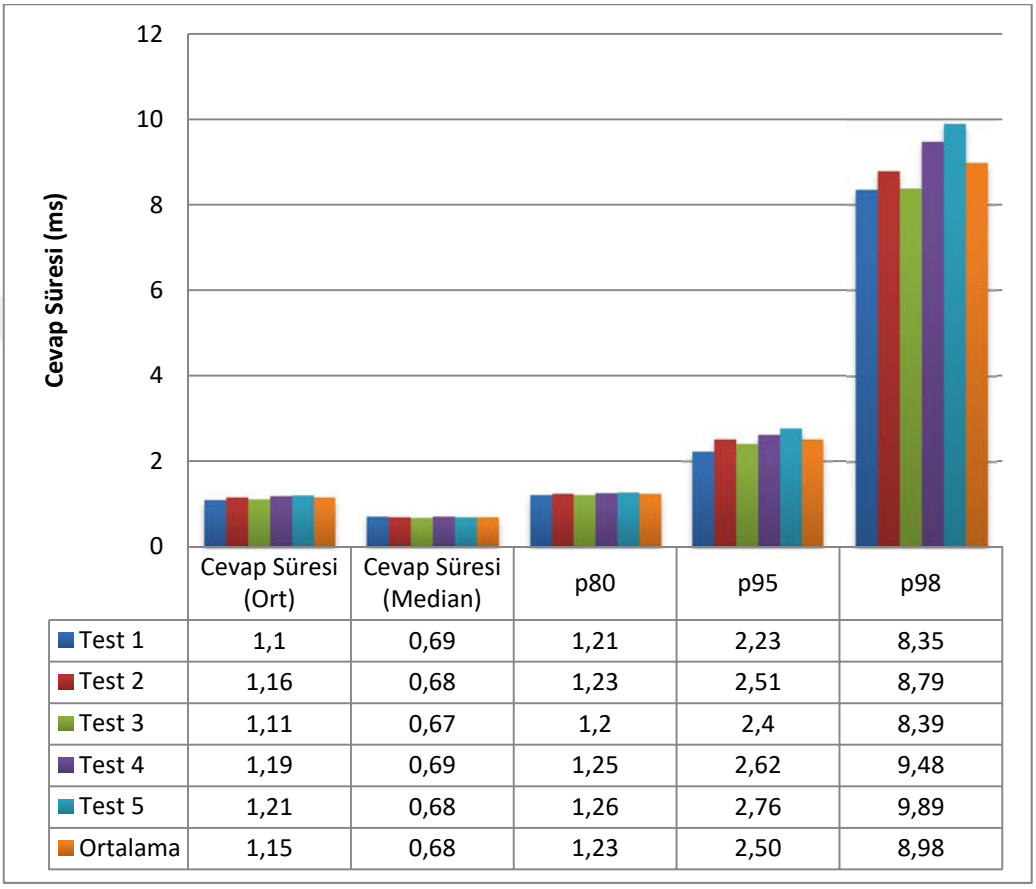
Şekil 4.13'te, k6 yük testi uygulamasının sonuçları gözlemlenebilir. Doğrusal olarak istek sayısı artışı uygulanarak, 1000 birimde sabitlenmiş ve cevap süresi sonuçları kaydedilmiştir.



Şekil 4.13: Node.js programlama dili cevap süresi grafiği (k6 yük testi)

Yapılan bu testte, kullanıcı sayısı 0'dan 1000'e doğrusal olarak arttırılırken, ortalama cevap süresi 0.5ms ile 1.8ms arasında olduğu, kullanıcı sayısı 1000'e ulaştıktan sonra 1 dakika boyunca devam ettirildiğinde ise, ortalama cevap süresinin 1ms ile 2ms aralığında değiştiği; servisin performansını daha detaylı olarak gösteren p95 cevap süresi metriği incelendiğinde ise, anlık olarak 30ms'e çıkmış olmasına rağmen, test boyunca 2ms – 8ms aralığında seyrettiği gözlemlenmiştir. Bu bağlamda uygulamanın performansını sürdürülebilir şekilde koruyabildiği yorumu yapılabilir.

Şekil 4.14'te Locust yük testi için cevap süresi grafiği bulunmaktadır. Aynı şartlarda 5 kere tekrarlanarak, ortalama cevap süresi, medyan cevap süresi, minimum cevap süresi, maksimum cevap süresi, p80 orantılı cevap süresi, p95 orantılı cevap süresi ve p98 orantılı cevap süresi kaydedilmiştir.



	Cevap Süresi (Ort)	Cevap Süresi (Median)	p80	p95	p98
Test 1	1,1	0,69	1,21	2,23	8,35
Test 2	1,16	0,68	1,23	2,51	8,79
Test 3	1,11	0,67	1,2	2,4	8,39
Test 4	1,19	0,69	1,25	2,62	9,48
Test 5	1,21	0,68	1,26	2,76	9,89
Ortalama	1,15	0,68	1,23	2,50	8,98

Şekil 4.14: Node.js programlama dili cevap süresi grafiği (k6 yük testi)

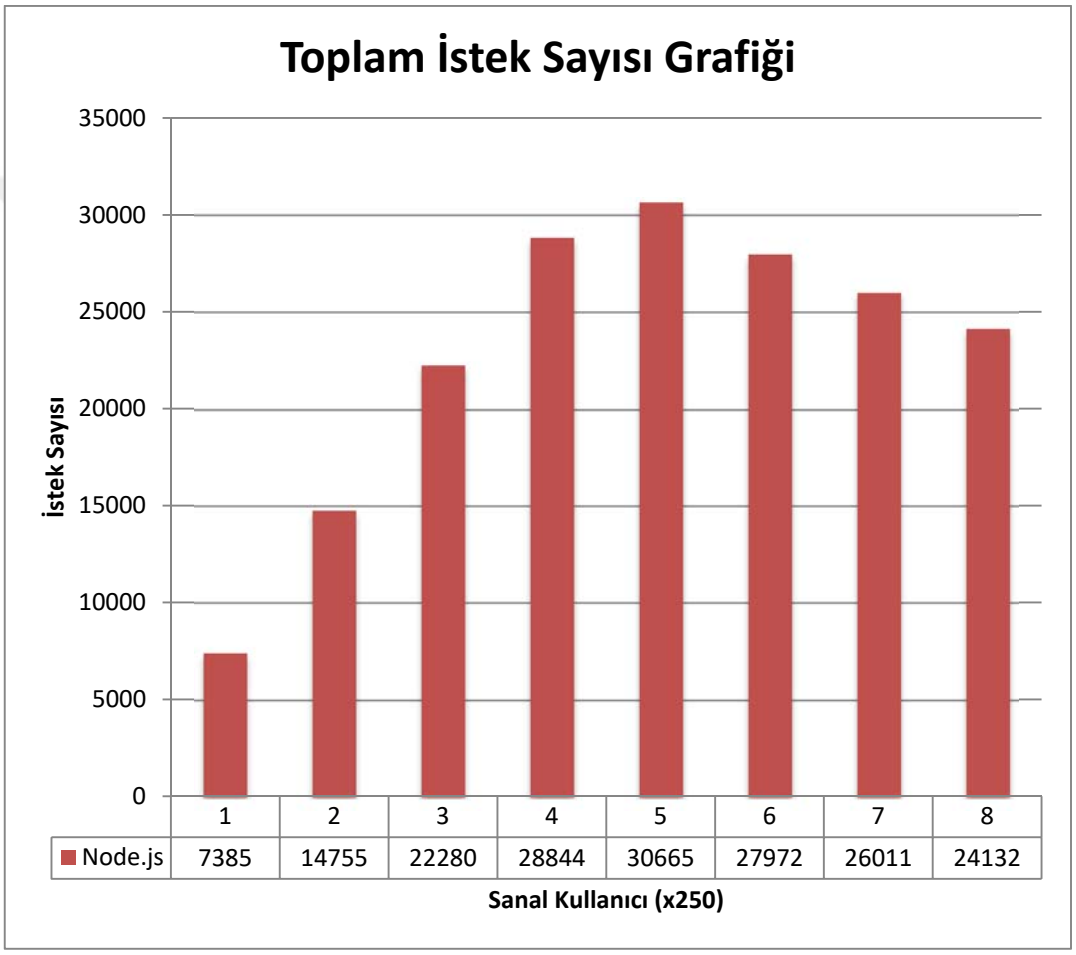
Bu grafiğe bakıldığında,

- tüm testler için ortalama cevap sürelerinin birbirine oldukça yakın olduğu ve 1,1ms - 1,21ms aralığında gerçekleştiği,
- tüm testler için medyan cevap sürelerinin birbirine oldukça yakın olduğu ve 0,67ms - 0,69ms aralığında gerçekleştiği,
- tüm testler için p80 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 1,21ms - 1,26ms aralığında gerçekleştiği,
- tüm testler için p95 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 2,23ms - 2,76ms aralığında gerçekleştiği,

- tüm testler için p98 orantılı cevap sürelerinin birbirine oldukça yakın olduğu ve 8,35ms - 9,89ms aralığında gerçekleştiği,

görülmektedir.

Şekil 4.15'te Locust yük testi ile uygulamanın cevaplayabildiği toplam istek sayısının, istek yapan istemci sayısına göre değişimi kaydedilmiştir.

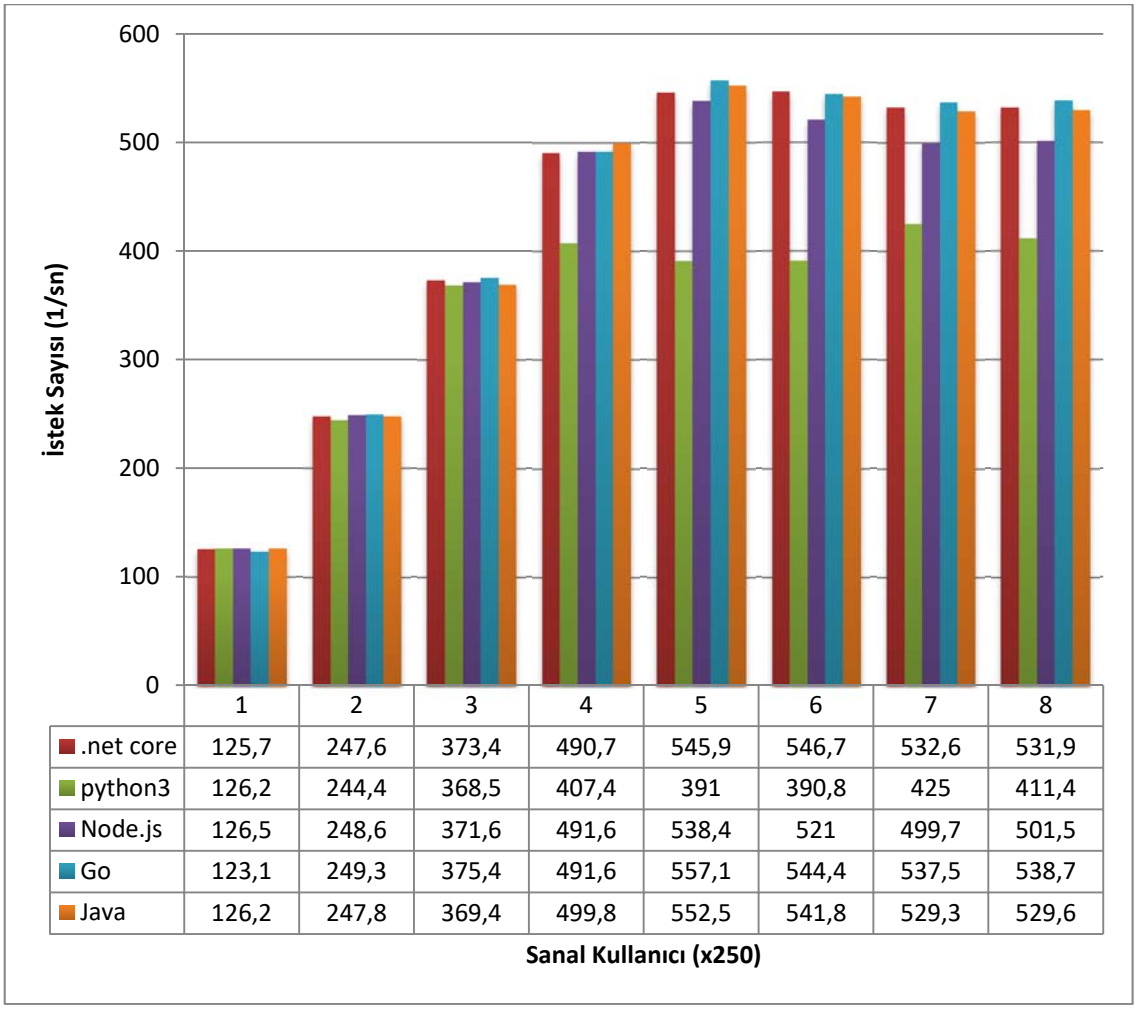


Şekil 4.15: Node.js programlama dili istek sayısı grafiği (Locust yük testi)

Testte, Node.js uygulamasının, 1250 istemciye kadar verebildiği başarılı istek sayısını arttırdığı, 1250 istemci sonrası ise, cevap sayısının artmadığı ve hatta azaldığı gözlemlenmiştir. Uygulamanın, kullanılan donanım ile optimum noktanın 1250 kullanıcı ile yapılan testte, toplam 30665 isteğe cevap vererek sağladığı gözlemlenmiştir.

4.2. PERFORMANS KARŞILAŞTIRMALARI

Şekil 4.16'da, 250 birimlik sanal kullanıcı artışına bağlı olarak, farklı programlama diliyle tasarlanan uygulamaların birim saniyedeki cevaplayabildiği istek sayısı değişimleri incelenebilir.



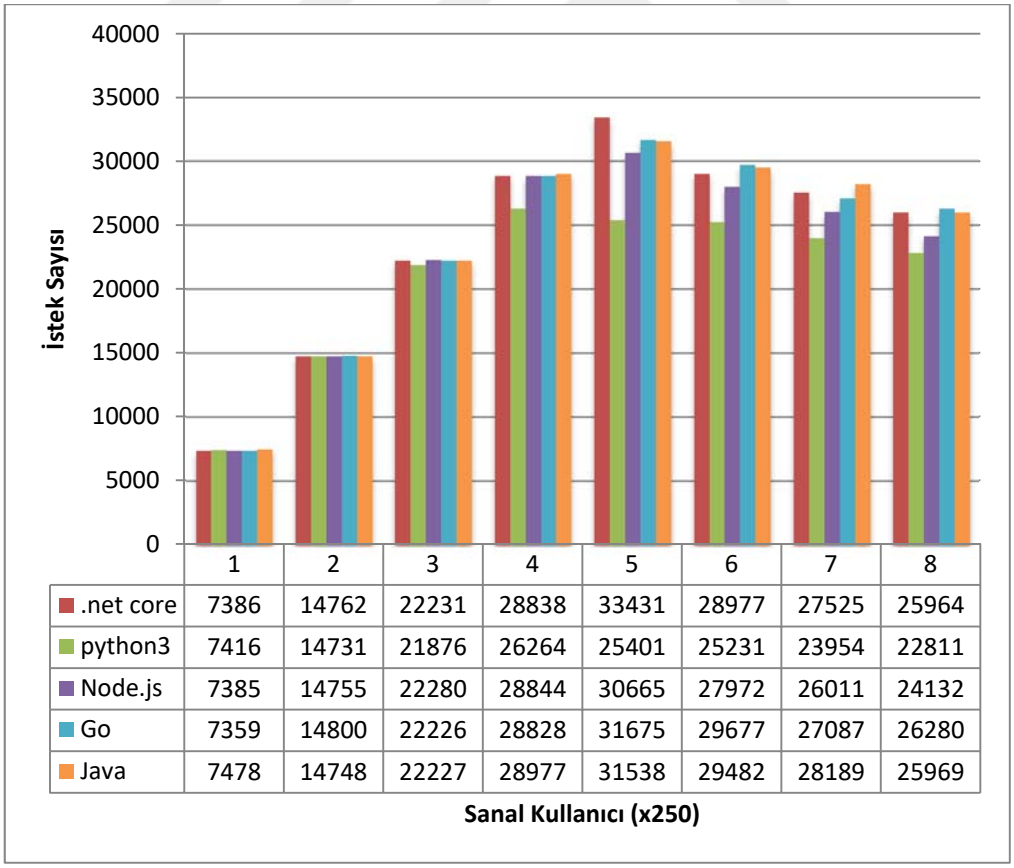
	1	2	3	4	5	6	7	8
.net core	125,7	247,6	373,4	490,7	545,9	546,7	532,6	531,9
python3	126,2	244,4	368,5	407,4	391	390,8	425	411,4
Node.js	126,5	248,6	371,6	491,6	538,4	521	499,7	501,5
Go	123,1	249,3	375,4	491,6	557,1	544,4	537,5	538,7
Java	126,2	247,8	369,4	499,8	552,5	541,8	529,3	529,6

Şekil 4.16: Maksimum istek sayısı grafiği (Locust)

Tüm platformlarda görülen ortak sonuç, kullanıcı sayısı arttıkça, belli bir noktaya kadar saniyede yapılabilen istek sayısı atmakla beraber, belli bir noktadan sonra bu değer sabit kalmakta, hatta bazı durumlarda düşmektedir. Bu noktalar platformun bu donanım üzerinde optimum kaldırabileceği maksimum yükü göstermektedir. Farklı platformlar farklı değerlerde limite ulaşmaktadır.

Şekilde de görüldüğü gibi Python platformu, 1000 kullanıcı ile yapılan yük testinde saniyede 407 isteğe cevap verebilmiş ve sonrasında istek yapan kullanıcı sayısı artmasına rağmen, saniyede verilen cevap sayısı artmamış, ortalama olarak sabit kalabilmiştir. Bu bakımdan performans bakış açısı ile Python en kötü performansı göstermiştir. Diğer diller genel olarak 1250 kullanıcı ile yapılan teste kadar, saniyede yapılan istek sayısına artan performanslı cevaplar vermiş, bu noktadan sonra genel olarak cevap performansları sabit kalmıştır. 1500, 1750 ve 2000 kullanıcı ile yapılan testlerde, saniyede verilen cevap sayıları, 1250 kullanıcı ile yapılan testlerden daha yüksek olmamıştır. Sonuç olarak kullanıcı sayısının arttırıldığı bu testte, Python saniyede 407 isteğe cevap vererek en kötü performansı göstermiş, diğer diller ortalama saniyede 550 isteğe cevap vererek benzer performans göstermiştir.

Şekil 4.17'de, 250 birimlik sanal kullanıcı artışına bağlı olarak, farklı programlama diliyle tasarlanan uygulamaların, bir dakika boyunca koşulan yük testleri süresince cevap verebildikleri toplam istek sayısı değişimleri incelenebilir.

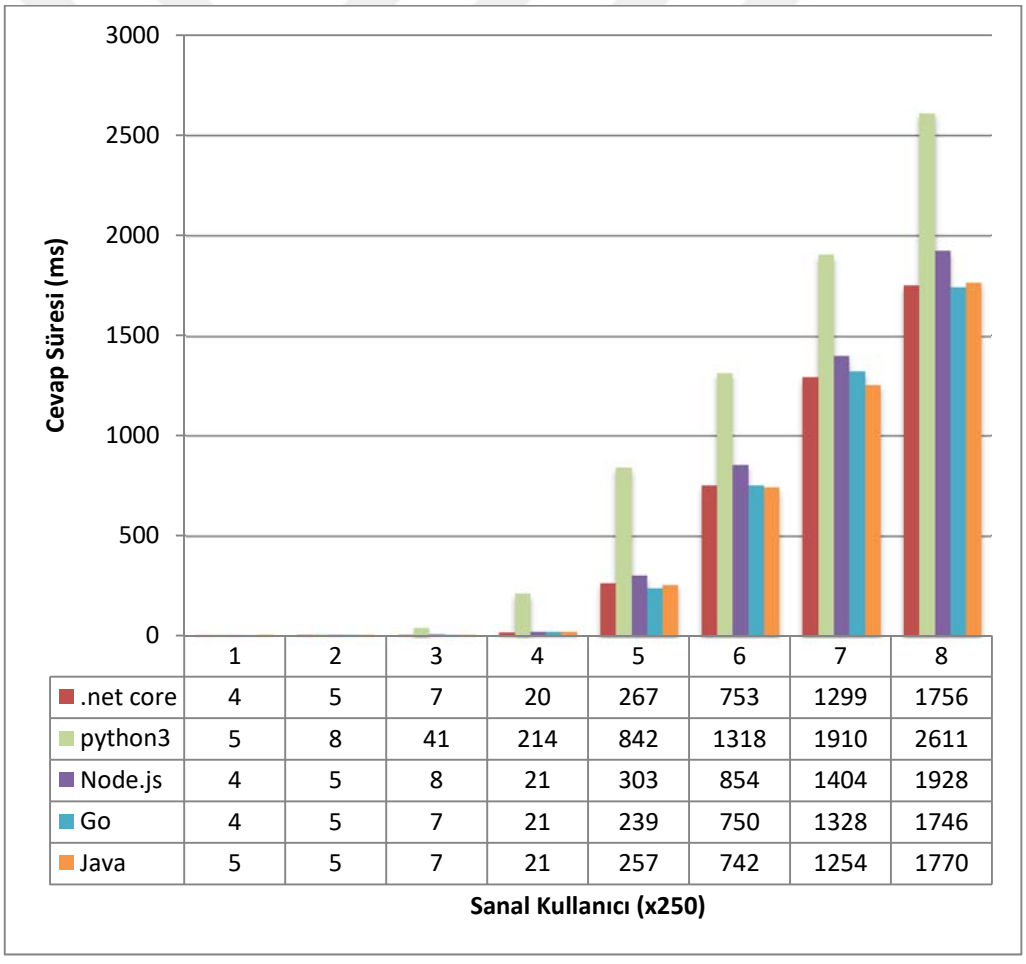


	1	2	3	4	5	6	7	8
.net core	7386	14762	22231	28838	33431	28977	27525	25964
python3	7416	14731	21876	26264	25401	25231	23954	22811
Node.js	7385	14755	22280	28844	30665	27972	26011	24132
Go	7359	14800	22226	28828	31675	29677	27087	26280
Java	7478	14748	22227	28977	31538	29482	28189	25969

Şekil 4.17: Toplam istek sayısı grafiği (Locust)

Yapılan bu testte, bir önceki testte elde edilen sonuçlara paralel sonuçlar alınmıştır. Bir dakika boyunca farklı sayıda sanal kullanıcılar ile koşulan testlerde, platform bazlı değişmek üzere, toplam yapılabilen istek sayısı belli bir kullanıcı sayısına kadar artabilmiş, belli bir noktadan sonra limite ulaşıp sabit kalmıştır.

Bir önceki teste benzer şekilde en kötü performans, 1000 kullanıcı ile yapılan testte toplam 28844 isteğe cevap verebilen Python'a aittir. En iyi performansı .Net Core, 1250 kullanıcı ile yapılan testte 33431 isteğe cevap vererek elde etmiştir. Ancak Node.js, Java ve Go dilleri ile yapılan uygulamalar da bu değerlere çok yakın performans göstermiştir. Şekil 4.18'de, 250 birimlik sanal kullanıcı artışına bağlı olarak, farklı programlama diliyle tasarlanan uygulamaların ortalama cevap süresi değişimleri incelenebilir.



	1	2	3	4	5	6	7	8
.net core	4	5	7	20	267	753	1299	1756
python3	5	8	41	214	842	1318	1910	2611
Node.js	4	5	8	21	303	854	1404	1928
Go	4	5	7	21	239	750	1328	1746
Java	5	5	7	21	257	742	1254	1770

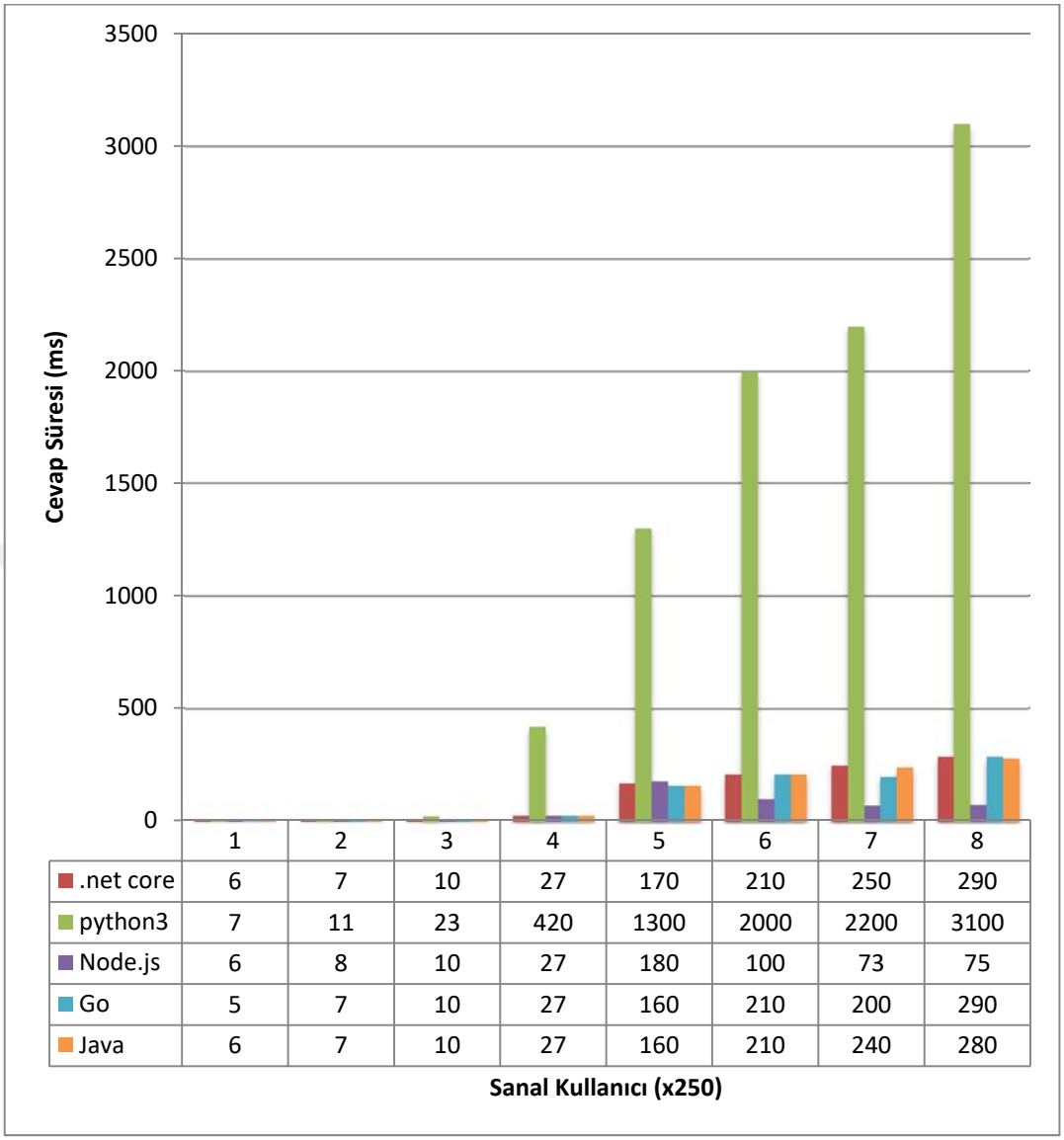
Şekil 4.18: Ortalama cevap süresi grafiği (Locust)

Ortalama cevap sürelerine baktığımızda 250 kullanıcı ile yapılan testte tüm platformlar 5 ms altında cevap vererek yüksek performans göstermiştir. Sonrasında 750 kullanıcı ile yapılan test ile başlayarak Python uygulaması, diğer diller ile geliştirilen uygulamalara göre daha geride kalmaya başlamaktadır. Ortalama olarak Python uygulaması 750 kullanıcı üzeri yapılan testlerde diğer platformlara göre %30 ile %60 oranında daha geç ortalama cevap süresi ile sonuç dönebilmiştir. Bunun haricinde diğer diller yaklaşık olarak birbirine yakın ortalama cevap süreleri ile performans göstermekle beraber, en iyi ortalama cevap süresi performansını tüm kullanıcı testlerini .Net Core platformu sağlamıştır.

Ortalama cevap süreleri web sistemleri performanslarını genel olarak gözlemlemek için kullanılan bir istatistiki değerdir. Fakat web uygulama performansları ölçümlenirken, ortalama değerler yanında p75, p80, p95 ve p99 gibi yüzdelik istatistikî değerlerin incelenmesi gerekmektedir. Örneğin p99, uygulamanın gelen tüm isteklerin %99'una ne kadar sürede cevap verdiğini gösteren bir değeri temsil eder. Bu değerler kullanarak ortalama değerin gözden kaçırdığı, aykırı değerlerin görülmesini de sağlar.

Özellikle web uygulamaları ve web programlama arayüzleri performanslarını ortalama cevap süreleri ile ölçümlemenin yanında yüzdelik performanslarını da ölçümlenmesi gerekmektedir. Aşağıdaki testlerde uygulamaların yüzdesel olarak yapılan ölçümlemelerde gösterdiği performansları karşılaştırılmaktadır.

Şekil 4.19'da, 250 birimlik sanal kullanıcı artışına bağlı olarak, farklı programlama diliyle tasarlanan uygulamaların p80 dilimindeki cevap süresi değişimleri incelenebilir. Bu test kapsamında uygulamanın cevap verebildiği istek sayısı değil, fakat uygulamanın isteklere ne kadar hızlı cevap verebildiği ölçülmüştür ve birbirleri arasında kıyaslanmıştır.

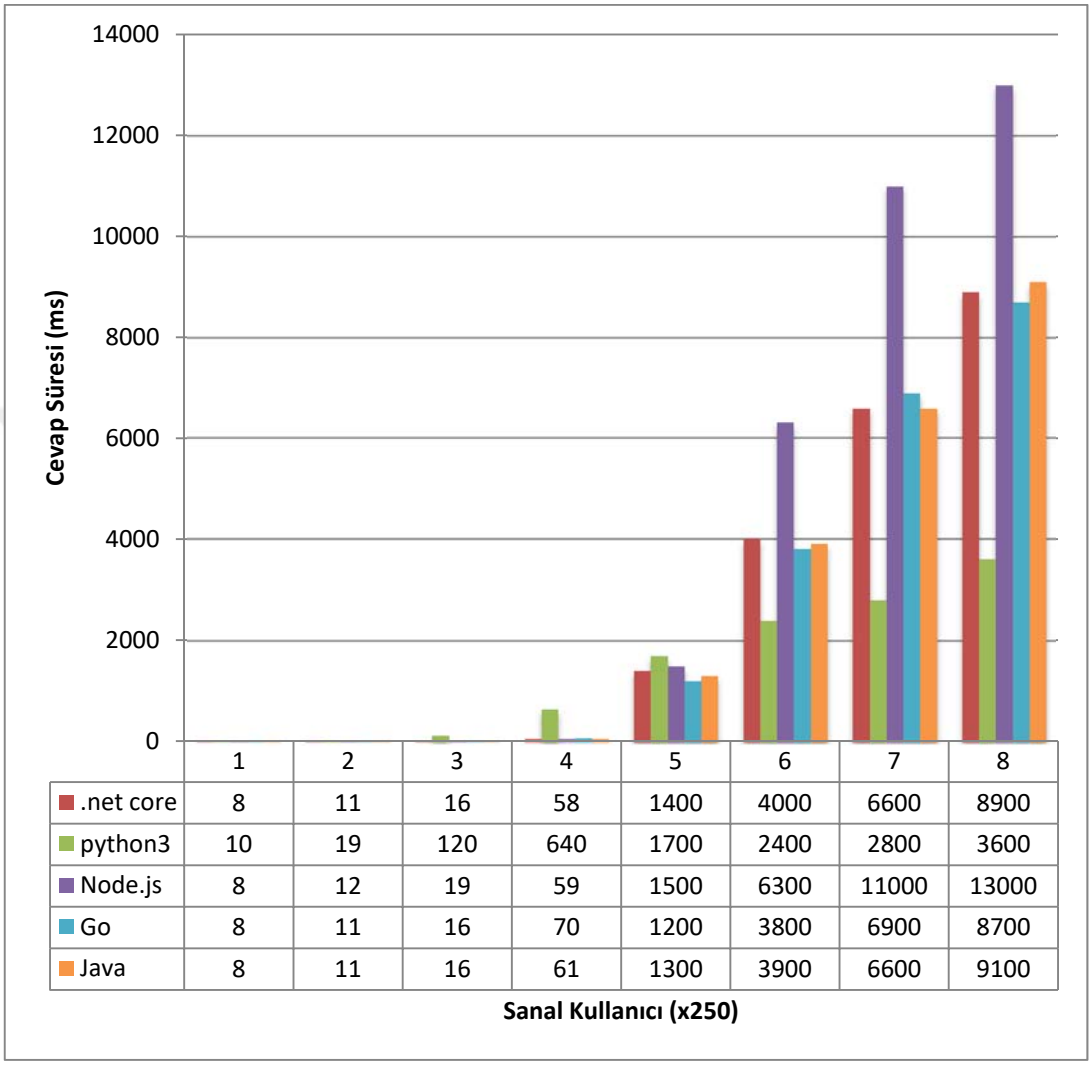


	1	2	3	4	5	6	7	8
.net core	6	7	10	27	170	210	250	290
python3	7	11	23	420	1300	2000	2200	3100
Node.js	6	8	10	27	180	100	73	75
Go	5	7	10	27	160	210	200	290
Java	6	7	10	27	160	210	240	280

Şekil 4.19: Cevap süresi grafiği - p80 (Locust)

Bu grafiğe bakıldığında, Python uygulamasının 1000 kullanıcı ve sonrasında p80'lik yüzdelik dilimde gelen isteklere ancak 420ms zaman diliminde cevap verebildiği görülmektedir. Gelen tüm isteklerin %80'ine ancak 420ms'de cevap veren Python uygulaması, 1250, 1500, 1750 ve 2000 kullanıcı ile yapılan testlerde de en kötü performansı gösteren uygulama olmuştur. Diğer taraftan Node.js uygulaması genel olarak p80'lik yüzdelik dilimde hızlı cevap süreleri noktasında en başarılı performans gösteren uygulama olmuştur. Java, .Net Core ve Go uygulamalarının da birbirlerine oldukça yakın değerlerde gelen isteklere cevap verebildikleri görülmektedir.

Şekil 4.20'de, 250 birimlik sanal kullanıcı artışına bağlı olarak, farklı programlama diliyle tasarlanan uygulamaların p95 dilimindeki cevap süresi değişimleri incelenebilir.



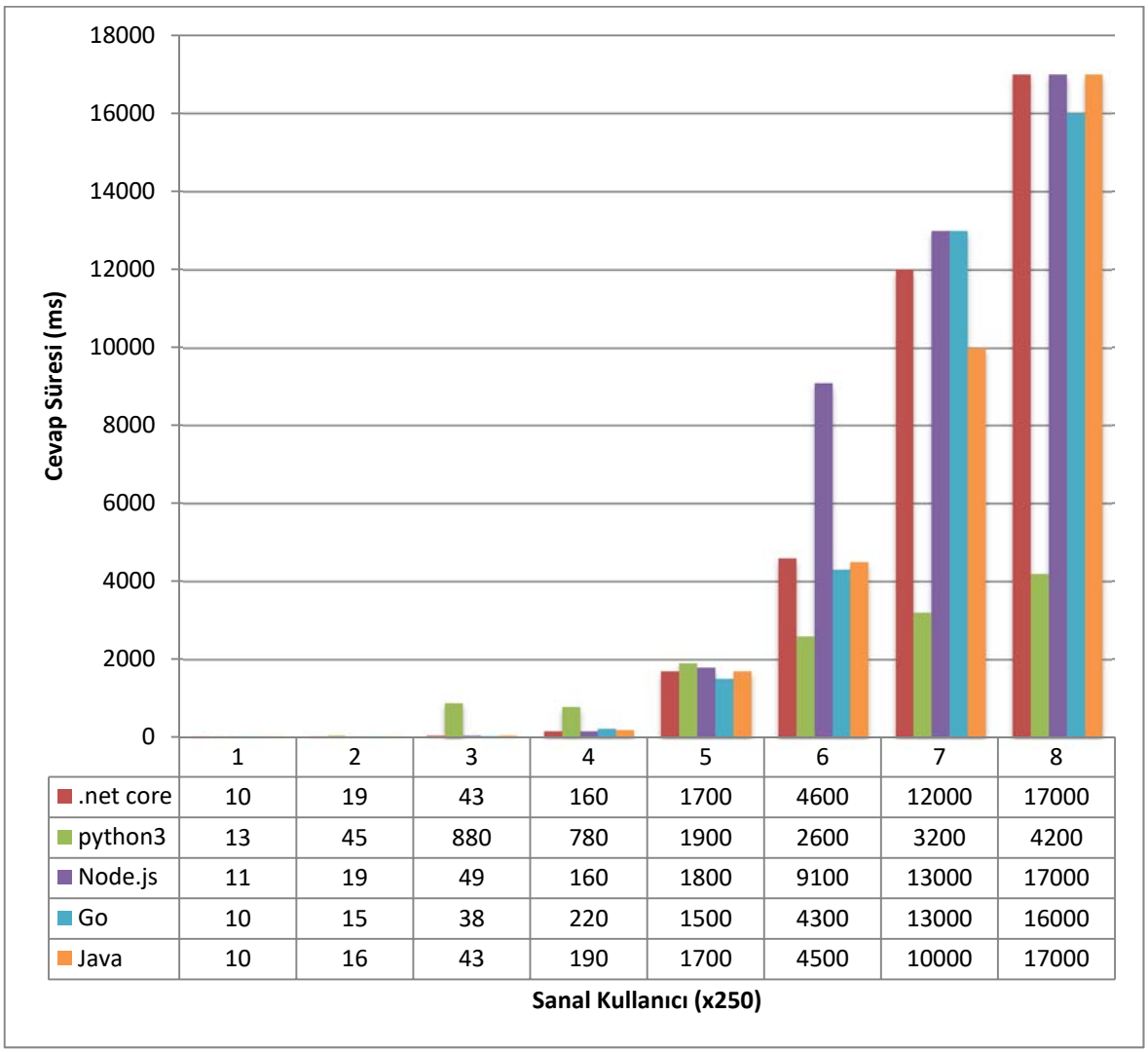
	1	2	3	4	5	6	7	8
.net core	8	11	16	58	1400	4000	6600	8900
python3	10	19	120	640	1700	2400	2800	3600
Node.js	8	12	19	59	1500	6300	11000	13000
Go	8	11	16	70	1200	3800	6900	8700
Java	8	11	16	61	1300	3900	6600	9100

Şekil 4.20: Cevap süresi grafiği - p95 (Locust)

Bu grafiğe bakıldığında, Python uygulamasının 1000 kullanıcı ile yapılan testte 640ms zaman diliminde cevap vererek en kötü performansı gösterdiği görülmektedir. 1250 kullanıcı ile yapılan testte ise Node.js uygulamasının gelen isteklerin yüzde 95'ine 6300ms içerisinde cevap vererek en kötü performansı gösterdiği gözlemlenmiştir. Node.js uygulaması aynı şekilde 1500, 1750 ve 2000 sanal kullanıcı ile yapılan testlerde de gelen isteklere en yavaş cevap dönebilen uygulama olmuştur. p80'lik yüzdelik dilimde en başarılı cevapları verebilen Node.js uygulamasının p95'lik yüzdelik dilimde böyle bir sonuç elde etmesi, uygulamaları test ederken farklı yüzdelik dilimlerdeki sonuçların incelenmesinin ne kadar önemli olduğunu ortaya

koymuştur. Diğer taraftan Python uygulaması genel olarak p95'lik yüzdelik dilimde hızlı cevap süreleri noktasında diğer uygulamalara göre daha iyi performans göstermiş olsa da, tüm uygulamaların genel olarak kabul edilebilen sınırların üzerinde gelen isteklere cevap verdikleri görülmektedir.

Şekil 4.21'de, 250 birimlik sanal kullanıcı artışına bağlı olarak, farklı programlama diliyle tasarlanan uygulamaların p99 dilimindeki cevap süresi değişimleri incelenebilir.



	1	2	3	4	5	6	7	8
.net core	10	19	43	160	1700	4600	12000	17000
python3	13	45	880	780	1900	2600	3200	4200
Node.js	11	19	49	160	1800	9100	13000	17000
Go	10	15	38	220	1500	4300	13000	16000
Java	10	16	43	190	1700	4500	10000	17000

Şekil 4.21: Cevap süresi grafiği - p99 (Locust)

Bu grafiğe bakıldığında, Python uygulamasının 750 kullanıcı ile yapılan testte 880ms zaman diliminde cevap vererek en kötü performansı gösterdiğini görmekteyiz. 1000 kullanıcı ile yapılan testte de yine Python uygulamasının gelen isteklerin yüzde 99'una 780ms içerisinde

cevap vererek en kötü performansı gösterdiği gözlemlenmiştir. Bu testlerde diğer uygulamalar birbirlerine yakın performans göstermişlerdir. 1250 kullanıcı ile yapılan testte hemen hemen tüm uygulamalar birbirlerine yakın performans göstermişlerdir. 1500, 1750 ve 2000 kullanıcı ile yapılan testlerde Python uygulaması göreceli olarak diğer uygulamalardan daha iyi performans gösterse de, tüm uygulamaların gelen isteklere kabul edilebilir cevap sürelerinin çok üzerinde sürelerde cevap verebildiği gözlemlenmiştir.

5. TARTIŞMA VE SONUÇ

Bu tez çalışmasında farklı programlama dilleri kullanarak prototip WebAPI uygulamaları geliştirilmiş, geliştirilen uygulamaların farklı yükler altında göstereceği performansları kayıt altına alınıp, sonuçlar birbiri ile karşılaştırılmıştır. Bu karşılaştırma işlemleri C#, Java, Go, Python ve Node.js programlama dilleri kullanılarak gerçekleştirilmiştir. Uygun bir karşılaştırma yapılabilmesi için uygulamalar, programlama dillerinde ek bir kütüphane kullanmadan veya farklı ek bir iş yapılmadan, gelen istekleri karşılayarak, basit cevaplar dönecek şekilde tasarlanmıştır. Geliştirilen WebAPI uygulamaları, iki farklı yük testi uygulaması kullanılıp farklı yükler oluşturularak test edilmiştir. Test sonuçları kayıt altına alınarak sonrasında performans karşılaştırmalarında kullanılmıştır.

Yapılan ilk iki yük testi setinde k6 adlı uygulama; üçüncü yük testi setinde ise Locust isimli uygulama baz alınmıştır. Sanal kullanıcı altyapısı sağlayan bu uygulamalarda, bu sanal kullanıcıların davranışları kodlanarak uygulamalar yük altında test edilmiştir. Testler çeşitli kullanıcı sayılarıyla pek çok kez tekrar edilmiş ve performanslar her seferinde ayrı ayrı hesaplanmıştır.

k6 uygulaması ile gerçekleştirilen ilk yük testi seti sonucunda, en kararlı şekilde isteklere cevap verebilen uygulama Go ile geliştirilmiş WebAPI uygulaması olduğu gözlemlenmiştir. C# ve Java dilleri ile geliştirilen uygulamalarında yakın bir performans sergiledikleri gözlemlenmiştir. Node.js ile yazılan WebAPI uygulaması da kararlı bir istek cevaplama performansı göstermesine karşı, ortalama cevap süreleri, diğer uygulamaların biraz gerisinde kalmıştır. En düşük performansı Python ile geliştirilen uygulama göstermiştir. Uygulamanın, test süresince oldukça kararsız bir ortalama cevap süresi ile birlikte isteklere cevap verdiği gözlemlenmiştir. k6 uygulaması ile gerçekleştirilen ikinci test seti sonrasında, alınan sonuçların ortalamaları incelendiğinde, en iyi ortalama cevap süresinin Go dili ile geliştirilen uygulamadan alınabildiği gözlemlenmiştir. Sonrasında C# ve Java dilleri ile geliştirilen uygulamalar birbirine çok yakın şekilde ikinci en iyi performansı sergilemiştir. Node.js uygulaması üçüncü en iyi ortalama cevap süresini verirken, Python uygulamasının en kötü ortalama cevap süresine sahip olan uygulama olduğu gözlemlenmiştir. Uygulamaların en hızlı cevap süreleri karşılaştırıldığında ise tersi bir durumun olduğu görülmüştür. Python ve Node.js ile geliştirilen WebAPI uygulamalarının en iyi, en hızlı cevap performansını gösterdiği gözlemlenmiştir.

Locust uygulaması ile yapılan testlerde, tüm geliştirilen WebAPI uygulamalarının 750 sanal kullanıcıya kadar yüksek performans gösterdikleri, hem ortalama cevap sürelerinin hem de cevaplanan toplam istek sayılarının birbirine çok yakın olduğu gözlemlenmiştir. 750 sanal kullanıcının üzerine çıkıldığında ise Python ile geliştirilen WebAPI uygulamasının hem ortalama cevap süresinde, hem de cevaplanan toplam istek sayısında bir düşüş başladığı gözlemlenmiştir. C#, Java, Go ve Node.js programlama dilleri ile geliştirilen uygulamalar ise optimum performanslarını 1250 sanal kullanıcı ile yapılan testte vermişlerdir. Bunun sonrasında 1500, 1750 ve 2000 sanal kullanıcı ile yapılan testlerde uygulamalar zorlanmaya başlamış, 60 saniyelik testlerde, kullanıcı sayısı ve yapılabilecek istek sayısı artmasına rağmen, uygulamaların verebildikleri toplam istek sayısında bir artış gözlenmemiştir.

Yapılan tüm testler sonucunda, özellikle daha fazla yük altında en iyi performansı Go programlama dili ile geliştirilmiş WebAPI uygulamasının gösterdiği görülmüştür. C# ve Java dilleri ile geliştirilen uygulamaların da tüm testlerde birbirine yakın performanslar sergilediği, hem düşük hem yüksek yük altında hep kararlı bir şekilde çalıştığı tespit edilmiştir. Node.js ve Python ile geliştirilen uygulamaların yük altında daha düşük performans gösterirken, düşük yük altında yapılan testlerde ise yüksek sonuçlar gösterdiği gözlemlenmiştir. Alınan sonuçlar incelendiğinde, dillerin tasarım prensipleriyle oldukça tutarlı çıktıların elde edildiği görülebilir. Python ve Node.js derlenen diller olmayıp, yorumlanan diller kategorisinde yer almaktadırlar. Bu durum, uygulamaların yük altında performanslarının daha düşük düzeyde olmasının önemli nedenlerinden biri olarak sayılabilir. C# ve Java dilleri ara dillere derlenip, anında derleme yöntemi ile makine diline çevrilmektedir. Bu bağlamda özellikle Python uygulamasından yüksek performans göstermeleri beklenti dahilindedir. Go dili ile geliştirilen uygulama ise, Go derleyicisi tarafından direkt olarak ikili makine koduna çevrilir. Bu da Go dili ile geliştirilen uygulamaların, yük altında da yüksek performans verebilmesini mümkün kılmaktadır. Bu durum yapılan testlerde de gözlemlenmiştir.

Bu tez çalışması kapsamında, farklı programlama dilleri ile geliştirilen WebAPI uygulamalarının, diğer etmenlerden bağımsız olarak performansları birbiri ile kıyaslanmıştır. Performansa etki edebilecek, uygulama mimarisi, uygulama yayılma stratejileri, web uygulama sunucuları gibi etmenler de bulunmaktadır. Bu çalışmanın devamında, diğer etmenlerin de uygulamaların içerisine eklenerek, uygulama performanslarına olan etkilerinin incelenmesi planlanmaktadır.

KAYNAKLAR

[1]. Höfner, P. ve Lautenbacher, F., 2008. Algebraic structure of web services. Electronic Notes in Theoretical Computer Science. 200, 171-187.

[2]. Fahat, M., Moalla, N. ve Ourzout Y., 2015. Dynamic execution of a business process via web service selection and orchestration. Procedia Computer Science. 51, 1655-1664.

[3]. De Renzis, A., Garriga, M., Flores, A. ve Cechich, A., 2016. Case-based reasoning for web service discovery and selection. Electronic Notes in Theoretical Computer Science. 321, 89-112.

[4]. Mikkonen, T. ve Salminen, A., 2012. Implementing mobile mashware architecture: downloadable components as on-demand services. Procedia Computer Science. 10, 553-560.

[5]. El-Kafrawy, P., Elabd, E. ve Fathi, H., 2015. A trustworthy reputation approach for web service discovery. Procedia Computer Science. 65, 572-581.

[6]. Kaouan, M., Bouchiha, D. ve Benslimane, S. M., 2015. Shared-repository based approach for storing and discovering web. Procedia Computer Science. 73, 56-65.

[7]. Yue, H. ve Tao, X., 2012. Web services security problem in service-oriented architecture. Physics Procedia. 24, 1635-1641.

[8]. Sun, J., Sun, Z., Li, Y. ve Zhao., S., 2012. A strategic model of trust management in web services. Physics Procedia. 24, 1560-1566.

[9]. Xiao-Cong, X., Xiang-Qun, W., Kai-Yao, F. ve Yi-Jiang Z., 2012. Grey relational analysis on factors of the quality of web service. Physics Procedia. 33, 1992-1998.

[10]. Herdiyanti, A., Adityaputri, A. N. ve Astuti, H. M., 2017. Understanding the quality gap of information technology services from the perspective of service provider and consumer. Procedia Computer Science. 124, 601-607.

[11]. Honamore, S. ve Rath, S. K., 2016. A web service reliability prediction using HMM and fuzzy logic models. Procedia Computer Science. 93, 886-892.

[12]. Singh, R. P. ve Pattanaik, K. K., 2013. An approach to composite QoS parameter based web service selection. Procedia Computer Science. 19, 470-477.

[13]. Mezni, H., Chainbi, W. ve Ghedira, K., 2017. AWS-Policy: an extension for autonomic web service description. Procedia Computer Science. 10, 915-920.

[14]. Casado, R., Tuya, J. ve Younas, M., 2012. A family of test criteria for web services transactions. Procedia Computer Science. 10, 880-887.

[15]. Çiçek, A. N, 2009. Restful web servisleri ile e-sağlık sistemleri gerçekleştirimi. TOBB Ekonomi ve Teknoloji Üniversitesi Fen Bilimleri Enstitüsü.

[16]. Alonso, F. S., 2015. Development of a restful API – HATEOAS & driven API. Turku University of Applied Sciences.

[17]. Olsson, R., 2014. Applying REST principles on local client-side APIs. Master's Thesis at CSC.

[18]. Dudhe, A. ve Sherekar, S. S., 2014. Performance analysis of SOAP and RESTful mobile web services in cloud environment. International Journal of Computer Applications. 975, 8887.

[19]. Browne, S., Dongarra, J., Garner, N., Ho, G. ve Mucci, P., 2000. A Portable Programming Interface for Performance Evaluation on Modern Processors, The International Journal of High Performance Computing Applications, 14(3), 189-204.

[20]. Polychronakis, M., Markatos, E. P., Anagnostakis, K.G. ve Oslebo, A., 2004. Design of an application programming interface for IP network monitoring, IEEE/IFIP Network Operations and Management Symposium, Seoul, South Korea

[21]. Van der Linden, S., Rabe, A., Held, M., Jakimow, B., Leitão, P. J., Okujeni, A., Schwieder, M., Suess, S. ve Hostert, P., 2015. The EnMAP-Box - A Toolbox and Application Programming Interface for EnMAP Data Processing, Remote Sensing, 7(9), 11249-11266.

[22]. Zambelli, P., Gebbert, S. ve Ciolli, M., 2013. Pygrass: An Object Oriented Python Application Programming Interface (API) for Geographic Resources Analysis Support System (GRASS) Geographic Information System (GIS), International Journal of Geo-Information, 2(1), 201-219.

[23]. Buck, B. ve Hollingsworth, J. K., 2000. An API For Runtime Code Patching, The International Journal of High Performance Computing Applications, 14(4), 317-329.

[24]. Bornstein, B. J., Keating, S. M., Jouraku, A. ve Hucka, M., 2008. LibSBML: an API Library for SBML, Bioinformatics, 24(6), 880–881.

[25]. Sparks, E. R., Talwalkar, A., Smith, V., Kottalam, J., Pan, X., Gonzalez, J., Franklin, M. J., Jordan, M. I. ve Kraska, T., 2013. MLI: An API for Distributed Machine Learning, 2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA.

[26]. Bowen, J. F., Mathkar, A. R., Mathur, R., Syed, S. M. A., Weimer, T. W., Bennett, J. E., Braganza, C. W. ve Dwivedi, T., 2002. US7401338B1 numaralı patent.

[27]. Hartmaier, P. J., Gossman, W. E., 1996. US5978672A numaralı patent.

[28]. Desai, A. A., Fussell, M. W., Kimball, A.E., Brundage, M.L., Dubinets, S., Pfleiger, T.F., 2003. US7383255B2 numaralı patent.

[29]. Bloesch, A. ve Rajagopal, R., 2003. US7293254B2 numaralı patent.

[30]. Eisler, C. G. ve Engstrom, G. E., 1997. US6128713A numaralı patent.

[31]. Strom, D. J. ve Zeliger, O., 2001. US7010796B1 numaralı patent.

[32]. Evans, C. S., Andrews, M. J., Sharma, O. K., Veres, J. E. ve Thornton, J. M., 1999. US7116310B1 numaralı patent.

[33]. Boillot, M. A., 2006. US8312479B2 numaralı patent.

EKLER

EK 1. k6 Yük Testi Uygulaması Kaynak Kodları

EK 1.1. C# Programlama Dili için Uygulanan k6 Yük Testi

```
import http from "k6/http";
import { check, sleep } from "k6";
export let options = {
  summaryTrendStats: ["avg","med","min","max","p(80)","p(90)","p(95)","p(98)"],
  stages: [
    { duration: "10s", target: 20 },
    { duration: "10s", target: 40 },
    { duration: "10s", target: 60 },
    { duration: "10s", target: 80 },
    { duration: "10s", target: 100 },
    { duration: "60s"},
    { duration: "10s", target: 1 },
  ]
};

export default function() {
  let params = {
    timeout: 10000
  };

  let result = http.get("http://127.0.0.1:5000/api/ping", params);
  check(result, {
    "status was 200": (r) => r.status == 200
  });
  sleep(0.1);};
```

EK 1.2. Java Programlama Dili için Uygulanan k6 Yük Testi

```
import http from "k6/http";
import { check, sleep } from "k6";

export let options = {
  summaryTrendStats: ["avg","med","min","max","p(80)","p(90)","p(95)","p(98)"],
  stages: [
     { duration: "10s", target: 20 },
     { duration: "10s", target: 40 },
     { duration: "10s", target: 60 },
     { duration: "10s", target: 80 },
     { duration: "10s", target: 100 },
     { duration: "60s"},
     { duration: "10s", target: 1 },
   ]
 };

export default function() {
  let params = {
    timeout: 10000
  };

  let result = http.get("http://localhost:5005/api/ping", params);
  check(result, {
    "status was 200": (r) => r.status == 200
  });
  sleep(0.1);
};
```

EK 1.3. Go Programlama Dili için Uygulanan k6 Yük Testi

```
import http from "k6/http";
import { check, sleep } from "k6";

export let options = {
  summaryTrendStats: ["avg","med","min","max","p(80)","p(90)","p(95)","p(98)"],
  stages: [
     { duration: "10s", target: 20 },
     { duration: "10s", target: 40 },
     { duration: "10s", target: 60 },
     { duration: "10s", target: 80 },
     { duration: "10s", target: 100 },
     { duration: "60s"},
     { duration: "10s", target: 1 },
   ]
 };

export default function() {
  let params = {
    timeout: 10000
  };

  let result = http.get("http://localhost:5001/api/ping", params);
  check(result, {
    "status was 200": (r) => r.status == 200
  });
  sleep(0.1);
};
```

EK 1.4. Python Programlama Dili için Uygulanan k6 Yük Testi

```
import http from "k6/http";
import { check, sleep } from "k6";

export let options = {
  summaryTrendStats: ["avg","med","min","max","p(80)","p(90)","p(95)","p(98)"],
  stages: [
     { duration: "10s", target: 20 },
     { duration: "10s", target: 40 },
     { duration: "10s", target: 60 },
     { duration: "10s", target: 80 },
     { duration: "10s", target: 100 },
     { duration: "60s"},
     { duration: "10s", target: 1 },
   ]
 };

export default function() {
  let params = {
    timeout: 10000
  };

  let result = http.get("http://127.0.0.1:5002/api/ping", params);
  check(result, {
    "status was 200": (r) => r.status == 200
  });
  sleep(0.1);
};
```

EK 1.5. Node.js Programlama Dili için Uygulanan k6 Yük Testi

```
import http from "k6/http";
import { check, sleep } from "k6";

export let options = {
  summaryTrendStats: ["avg","med","min","max","p(80)","p(90)","p(95)","p(98)"],
  stages: [
     { duration: "10s", target: 20 },
     { duration: "10s", target: 40 },
     { duration: "10s", target: 60 },
     { duration: "10s", target: 80 },
     { duration: "10s", target: 100 },
     { duration: "60s"},
     { duration: "10s", target: 1 },
   ]
 };

export default function() {
  let params = {
    timeout: 10000
  };

  let result = http.get("http://127.0.0.1:5006/api/ping", params);
  check(result, {
    "status was 200": (r) => r.status == 200
  });
  sleep(0.1);
};
```

EK 2. Locust Yük Testi Uygulaması Kaynak Kodları

```
import os

from locust import HttpLocust, TaskSet, task
from locust.clients import HttpSession



class UserTasks(TaskSet):
    # but it might be convenient to use the @task decorator
    @task
    def index(self):
        with self.locust.client.get(
                "/api/ping", catch_response=True, timeout=(2000,3000)) as response:
            if response is None or response.content is None:
                response.failure("No Response")
            elif response.content.decode("utf8") != "pong":
                response.failure("Got wrong response")
                print(response.content)
            else :
                response.success()



class WebsiteUser(HttpLocust):

    host = "http://127.0.0.1:5006"
    max_wait = 3000
    task_set = UserTasks
```

EK 3. C# Programlama Dili Rest API Uygulaması

```
using Microsoft.AspNetCore.Mvc;
namespace PerformanceBenchmarks.Dotnet.Controllers
{
   [Route("api/[controller]")]
   public class PingController : Controller
   {
      [HttpGet("")]
      public string Get(int id)
      {
         return "pong";
      }
   }
}
```

EK 4. Java Programlama Dili Rest API Uygulaması

```
package com.performancebenchmark.java.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
@RestController
@RequestMapping("/api/ping")
public class PingController {
   @RequestMapping(value = "",method = RequestMethod.GET)
   @ResponseBody
   public String get(){
      return "pong";
   }
}
```

EK 5. Go Programlama Dili Rest API Uygulaması

```
package main
import (
    "fmt"     "log"     "net/http"
    "github.com/gorilla/mux")
 func main() {
    router := mux.NewRouter()
    router.HandleFunc("/api/ping", Ping).Methods("GET")
    log.Fatal(http.ListenAndServe(":5001", router))}

func Ping(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "pong")}
```

EK 6. Python Programlama Dili Rest API Uygulaması

```
from flask import Flask, request
from flask_restful import Resource, Api
from json import dumps
from flask_jsonpify import jsonify
import logging
log = logging.getLogger('werkzeug')
log.setLevel(logging.ERROR)
app = Flask(__name__)
api = Api(app)
@app.route("/api/ping")
def ping():
   return "pong"
if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5002, threaded=True)
```

EK 7. Node.js Programlama Dili Rest API Uygulaması

```
import { Router } from 'express'
import ping from './ping'
const router = new Router()
router.use('/api/ping', ping)
export default router

import { Router } from 'express'
import { show } from './controller'
const router = new Router()
router.get('/', show)
export default router

export const show = ({ params }, res, next) =>
  res.status(200).send("pong")
```

ÖZGEÇMİŞ

Kişisel Bilgiler	
Adı Soyadı	Erdem KEMER
Doğum Yeri	Edirne
Doğum Tarihi	09.05.1984
Uyruğu	☑ T.C. ❒ Diğer:
Telefon	+90 549 761 98 44 / +353 89 708 66 34
E-Posta Adresi	erdemkemer@gmail.com
Web Adresi	

Eğitim Bilgileri	
Lisans	
Üniversite	İstanbul Üniversitesi
Fakülte	Mühendislik Fakültesi
Bölümü	Bilgisayar Mühendisliği
Mezuniyet Yılı	20.06.2005

Yüksek Lisans	
Üniversite	İstanbul Üniversitesi-Cerrahpaşa
Enstitü Adı	Lisansüstü Eğitim Enstitüsü
Anabilim Dalı	Bilgisayar Mühendisliği
Programı	Bilgisayar Mühendisliği

Makale ve Bildiriler
1. Erdem KEMER, Ruya SAMLI, **"Performance Comparison of Webapis in Different Platforms",** ICENS - 4th International Conference on Engineering and Natural Sciences, Kiev/UKRAYNA, 2-6 Mayıs 2018, Pages: 212