MİKROSERVİSLER İÇİN VERİMLİ YERLEŞTİRME ALTERNATİFLERİNİN TÜRETİLMESİ

DERIVING EFFICIENT DEPLOYMENT ALTERNATIVES FOR MICROSERVICES

IŞIL KARABEY AKSAKALLI

DOÇ. DR. AHMET BURAK CAN

Tez Danışmanı

Hacettepe Üniversitesi

Lisansüstü Eğitim - Öğretim ve Sınav Yönetmeliğinin

Bilgisayar Mühendisliği Anabilim Dalı için Öngördüğü

DOKTORA TEZİ olarak hazırlanmıştır.

2021

ÖZET

MİKROSERVİSLER İÇİN VERİMLİ YERLEŞTİRME ALTERNATİFLERİNİN TÜRETİLMESİ

Işıl KARABEY AKSAKALLI

Doktora, **Bilgisayar Mühendisliği Bölümü**
Tez Danışmanı: Doç. Dr. Ahmet Burak CAN
Eş danışman: Dr. Turgay ÇELİK
Haziran 2021, 147 sayfa

Mikroservis mimarileri gevşek bağlanmış, tek bir fonksiyonellik ile çalışarak bağımsız olarak konuşlandırılabilen ve birbirleri ile belirli bir arayüz üzerinden iletişimde olan modüler yazılım bileşenlerine dayalı popüler bir yaklaşımdır. Bağımsız yazılım bileşenleri sayesinde servislerin ölçeklenebilirliği, güncellenmesi ve bakımı oldukça kolaylaşmaktadır. Öte yandan mikroservis tabanlı uygulamaların bulut kaynaklarına yerleştirilmesi, tek parçalı (monolithic) uygulamalara göre daha karmaşık bir süreçtir. Tek parçalı bir uygulama, bir yük dengeleyici arkasında yer alan bir grup sunucuya tek seferde dağıtılabilirken, farklı servislerden oluşan bir mikroservis tabanlı uygulamada her servisin genellikle birden fazla çalışma zamanı örneği olduğu varsayıldığında, binlerce servisin yapılandırılması ve performanslı bir şekilde dağıtılması zorlu bir süreçtir. Az sayıda mikroservisin bulunduğu bir uygulama, sistemi iyi bilen bir uzman tarafından manuel bir şekilde dağıtılabilirken pratikte genellikle çok

sayıda mikroservisin yer aldığı uygulamalar görülmektedir. Artan servis sayısının bulut ortamındaki kaynaklara optimale yakın bir şekilde manuel olarak dağıtılması hem zaman alıcı bir hale gelmekte hem de kaynakların verimli kullanımını neredeyse imkansızlaştırmaktadır. Özellikle mikroservis tabanlı uygulamaların kaynak kullanımı ve performansı mikroservislerin mevcut kaynaklara uygun bir şekilde yerleştirilmesine bağlıdır. Mikroservislerin yerleştirilmesi için var olan Kubernetes, Docker, Apache Mesos gibi yönetim araçları, mikroservislerin çalışma zamanı verilerini dikkate almayıp minimum yetenek ile dağıtım yapmaktadır. Bununla birlikte literatürde mikroservisler için geliştirilen mevcut dağıtım yöntemleri üzerinden ek yaklaşımlar kullanılarak veya araç desteği geliştirilerek CPU iş yükü, G/Ç, ortalama yanıt süresi gibi kriterler açısından performans elde etmek amaçlanmaktadır. Fakat literatürde önerilen ve endüstride kullanılan mevcut dağıtım yöntemlerinde mikroservislerin sunuculara dağıtılması işlemi kullanıcıya bırakılmaktadır. Bu da çok sayıda servisten oluşan bir sistemde kullanıcının işini zorlaştırmakla birlikte, kabaca bir dağıtım yapılmasına sebep olarak, servislerin kısıtlı kaynaklara performanslı ve verimli bir şekilde dağıtılmasını engellemektedir.

Bu tez çalışmasında mikroservis mimarilerinin bulut ortamında verimli kullanılabilmesi için servislerin mevcut kaynaklar üzerine uygun yerleştirilmesi sorunu ele alınarak servislerin bulut sunucularına otomatik yerleştirilmesini sağlayan model güdümlü mimari ile geliştirilen bir yaklaşım önerilmektedir. Ayrıca bu yaklaşımı destekleyen bir araç ailesi geliştirilerek mikroservislerin minimum maliyet ile dağıtımı gerçekleştirilmektedir. Mikroservislerin etkin olarak kaynaklara dağıtımı, bulut kaynaklarının verimli ve daha yüksek performans ile kullanımını sağlayacaktır. Önerilen otomatik dağıtım yaklaşımının gelecekteki akademik çalışmalar için bir altyapı sağlamasının yanısıra, geliştirilen mikroservis dağıtım aracının endüstriyel uygulamalara önemli bir katkıda bulunacağı öngörülmektedir.

Anahtar Kelimeler: Model güdümlü mimari, Metamodelleme, Model dönüşümleri, Mikroservisler için verimli dağıtım yaklaşımları, Mikroservislerin otomatik dağıtımı için algoritmik tabanlı araç desteği

ABSTRACT

DERIVING EFFICIENT DEPLOYMENT ALTERNATIVES FOR MICROSERVICES

Işıl KARABEY AKSAKALLI

Doctor of Philosophy, **Computer Engineering Department**
Supervisor: Assoc. Prof. Dr. Ahmet Burak CAN
Co-supervisor: Dr. Turgay ÇELİK
June 2021, 147 pages

Microservice architecture is a popular approach that relies on modular software components that are loosely coupled, operate with a single functionality that can be independently deployed and communicate with each other via a well-defined interface. Thanks to independent software components, the scalability, update and maintenance of the services are easier. However, the deployment of microservice-based applications is more complicated than monolith applications. A monolith application can be deployed at one time on a group of services behind a load balancer. In a microservice-based application consisting of different services that have more than one runtime instance of each service, thousands of services need to be configured and deployed efficiently. While an application system with a small number of microservices can be manually deployed by an expert who knows the system well, a large

number of microservices are often used for the applications in practice. Efficient deployment of the increasing number of services to resources manually in the cloud environment becomes both time consuming and makes the efficient use of resources almost impossible. In particular, resource usage and performance of microservice-based applications depend on the efficient deployment of microservices to the available resources. Existing tools such as Kubernetes, Docker, Apache Mesos don't consider the runtime data of a microservices and deploy them with minimum ability. Also, in literature, it is foreseed to achieve performance in terms of criteria such as CPU workload, I/O, average response time by developing additional approaches or tool support over existing deployment tools. However, the configuration process for deploying microservices is left to a user in these methods. This situation makes it difficult for the user to work in a system consisting of many services, hence, it prevents the deployment of services to limited resources in an efficient manner.

In this thesis, an approach an approach developed using model driven architecture that provides automatic deployment of services to cloud servers is proposed. Furthermore, deployment of microservices at minimum cost has been performed by developing a tool that supports this approach. The approach and the tool support have been tested using an industrial case study. By effectively deploying microservices to resources, it will be possible to use the resources of the cloud environment efficiently and with higher performance. With the proposed automated deployment approach, it is foreseed to enable an infrastructure for the future academic studies, as well as a contribution to industrial applications with the developed microservice deployment tool.

Keywords: Model-driven architecture, Metamodeling, Model Transformations, Efficient deployment approach for microservices, Algorithmic based tool support for deploying microservices automatically

TEŞEKKÜR

Lisansüstü eğitim hayatımda bana yol gösteren, desteğini ve bilgilerini esirgemeyen tez danışmanlarım Sayın Doç. Dr. Ahmet Burak CAN ve Sayın Dr. Turgay ÇELİK'e,

Tez izleme aşamasında önerileriyle tezimin gelişmesine katkıda bulunan tez komite üyelerim Sayın Dr. Öğr. Üyesi Pelin ANGIN ve Sayın Dr. Öğr. Üyesi Fuat AKAL'a,

Makalelerimin yayınlanmasına katkıda bulunarak tezimin içeriğinin zenginleşmesini sağlayan Sayın Prof. Dr. Bedir TEKİNERDOĞAN'a

Tüm eğitim hayatım boyunca maddi ve manevi desteklerini hiçbir zaman esirgemeyen kıymetli aileme,

Son olarak sevgisini ve desteğini her zaman hissettiğim sevgili eşim Tugay AKSAKALLI'ya ve varlığı ile bana mutluluk veren biricik kızım Zeynep Duru'ya,

sonsuz şükranlarımı sunuyorum.

İÇİNDEKİLER

Sayfa

ÇİZELGELER DİZİNİ

Sayfa

ŞEKİLLER DİZİNİ

Sayfa

1. GİRİŞ

Mikroservisler gevşek olarak bağlanmış ve bağımsız olarak konuşlandırılabilen tek işlevselliğe sahip bileşenlerden oluşmaktadır. Her bir servis belirli bir işlevi yerine getirdiği için ihtiyaç doğrultusunda belirli servisler ölçeklenerek bulut bilişim kaynaklarının verimli kullanımı sağlanmaktadır. Bulut bilişimin kullanılmasıyla, nesne depolama, mesajlaşma kuyrukları ve yük dengeleme gibi farklı ihtiyaçlar için yönetilen hizmetler kullanılarak yazılım uygulamalarının uzak bulut sunucularına dağıtımı için gerekli altyapı geliştirme ve yönetim maliyetleri önemli ölçüde azaltılabilmektedir. Bu yüzden çoğu işletme, şirket içi altyapıları oluşturmak ve işletmek için çaba harcamak yerine işlevsel yetenekler geliştirmeye, ayrıca ölçeklenebilirlik ve hata toleransı gibi mimari konulara daha fazla odaklanmak için bulut bilişime yönelmektedir. Bulut platformları genellikle şirket içi sistemlerden daha düşük bir maliyete sahip olsa da, bulut kaynaklarının maliyeti göz ardı edilecek düzeyde değildir. Gereksiz kaynak kullanımını minimuma indirgemek ve maliyeti düşürmek için bulut kaynakları olabildiğince verimli şekilde kullanılmalıdır. Ancak bulut sunucuları üzerinde çalışan bir uygulamanın artan talebe göre ölçeklendirilmesi önem arz etmektedir [2].

Bulut bilişimin bir çok faydasına rağmen tek parçalı (monolithic) uygulamaların bulut sunucularına göçü pratikte zorlu bir süreçtir. Modülerlik eksikliği sebebiyle tek parçalı uygulamaların bir bütün halinde ölçeklenme ve dağıtım ihtiyacı bulut kaynaklarının gereğinden fazla kullanımına ve dolayısıyla kullanıcının daha fazla ücret ödemesine sebep olmaktadır. Tek parçalı uygulamalardan kaynaklı bu sınırlamaların üstesinden gelmek için uygulamaları birbirlerinden olabildiğince bağımsız, modüler ve tek bir işlevselliğe odaklı servislere ayıran mikroservis mimarileri önerilmektedir. Mikroservis mimarilerinin kullanımı, artan uygulama iş yükü ve performans dalgalanmaları durumlarında ölçeklemeyi büyük ölçüde desteklemektedir.

Mikroservis mimarileri ortaya çıkmadan önce katmanlı tek parçalı katmanlı mimariler (örneğin, Model View Controller-MVC) yaygın bir şekilde benimsenmekte idi. Ancak tek parçalı mimarilerde yazılım uygulamaları büyüdükçe ve tek bir kod tabanı üzerinde çok sayıda

geliştirici çalıştıkça mimari oldukça karmaşıklaşmakta ve bu durum da yazılım geliştirme yaşam döngüsünü yavaşlatmaya neden olmaktadır. Ölçeklenebilirliğin yanısıra geliştirme ve dağıtım güncellemelerinin kısa tutulması, dalgalanan trafik ile başa çıkılması, büyük veritabanlarının yönetimi, sisteme yeni özelliklerin eklenmesi ve var olan verilerin güncellenmesi gibi zorluklar tek parçalı mimarilerde yönetilemez hale gelmektedir. Bu yüzden son yıllarda modern yazılım geliştirmede tek parçalı yazılım mimarisine göre daha esnek ve hafif (lightweight) olarak bilinen mikroservis mimarilerine geçiş hızlı bir şekilde yaygınlaşmaktadır. Bulut tabanlı mikroservis mimarilerine uyum sağlayan Netflix [3], Amazon [4], Uber [5] ve Etsy [6] gibi birçok büyük kurumsal şirket bulunmaktadır. Günümüzde popülerliğini sürdüren Netflix'in tüm dünyada tarafından aynı anda ulaşılabilen ve hataya toleranslı altyapısı, mikroservis tabanlı mimari çözümü ile gerçekleşmektedir. Bu altyapıda 500'den fazla mikroservis tarafından işlenen yaklaşık iki milyar API uç nokta talebi işlenmektedir [7].

Bir mikroservis mimarisi, birbirlerinden bağımsız olarak tek bir fonksiyonelliğe sahip olan ve birbirleri ile veri alışverişinde bulunarak bir bütün uygulama şeklinde birbirini tamamlayan servislerden oluşmaktadır. Servislerin artan modülerliği mikroservis mimarileri yazılım uygulamaları için büyük ölçüde fayda sağlamaktadır. Öte yandan tek fonksiyonelliğe sahip küçük taneli servislerin etkileşimleri ve bu servislerin çalışma zamanı ortamında karmaşık yapılandırılması mikroservis tabanlı sistemlerin kullanımını karmaşıklaştırmaktadır. Mikroservisler arası iletişimler ölçeklenebilirlik açısından genellikle eşzamanlı olmayan (asynchronous) şekilde tasarlanmış olup karmaşık çağrı zincirlerine sahiptir. Örneğin, günde 5 milyon servis çağrısına sahip olan Netflix'te çağrıların %99.7'si diğer mikroservislerin basamaklı çağrıları ile sonuçlanmaktadır [7]. Benzer şekilde Amazon bir sayfayı yüklemek için çok sayıda mikroservis çağrısı ile başa çıkmaktadır [8]. Bununla birlikte bir mikroservis türü farklı kaynaklar üzerinde çalışan binlerce mikroservis örneğinden oluşabilmektedir. Bu mikroservis örneklerinin minimum maliyette verimli bir dağıtım yapılandırması ile yerleştirilmesi, bulut kaynaklarının daha verimli kullanımı için önem arz etmektedir. Bu yüzden çok sayıda mikroservisin dağıtım yapılandırması servisler arasındaki iletişim maliyeti ve hesaplama gücünün kullanımını büyük ölçüde etkilemektedir. Dağıtım konfigürasyonları genellikle sistemi iyi bilen bir grup uzman ile yürütülmektedir. Fakat çok büyük sistemlerde

mikroservis örneklerinin sayısı dramatik bir şekilde artmakta ve bu durum uzman değerlendirmesi ile verimli dağıtım alternatiflerinin türetilmesi zorlaşmakta ve algoritmik bir yaklaşım çözümüne göre daha uzun zaman almaktadır. Ayrıca bir mikroservis kümesini oluşturup çalıştırmaktan sorumlu bir yazılım ekibi, genellikle iletişim ve yürütme maliyetlerinin farkında olsa da, bir servis üzerinde yapılan bir değişikliğin diğer ilgili servisler üzerindeki etkisini pratikte tahmin etmeleri oldukça zordur [9]. Örneğin bir mikroserviste yapılan bir değişiklik, bu servisin çıktılarını tüketen diğer servisler üzerinde ek iletişim yükü oluşturabilmektedir. BU yüzden değişiklik ihtiyacı olan servisler üzerinde ölçeklenebilirlik problemi ortaya çıkmaktadır.

Mikroservislerin artan kullanımı ile birlikte konteynerlerin (container) popülerliği de artmaktadır [10]. Sanal makineler ve yazılım konteynerleri sanallaştırmayı sağlamakta ve mikroservislerin kullanıcı ihtiyaçlarına ve dağıtılan uygulamanın ihtiyacına bağlı olarak CPU, RAM ve bant genişliği gibi kaynak kısıtlarına göre yapılandırılarak servisler sanal sunuculara kolaylıkla dağıtılabilmektedir. Bir bulut ortamında, ana bilgisayar, sunucu ya da PM (Physical Machine) olarak bilinen fiziksel bir makine bir çok sanal makineye (Virtual Machine-VM) ev sahipliği yaparak bu VM'ler ile kaynaklarını paylaşmakta ve talebe bağlı olarak kullanıcı tarafından sınırsız gibi görünen kısıtlı kaynakları sanal makinelere sağlamaktadır. Sadece ihtiyaç halinde disk alanı, bellek ve CPU ayrımı yapılarak kaynakların gereksiz kullanımı önlenmektedir. Bu da bulut ortamında kaynak kullanımının iyileştirilmesi için önemli bir unsurdur.

Mikroservisler kullanıcı taleplerine uyum sağlamak ve yüksek kullanılabilirlik için genellikle ölçeklendirilmektedir. Bir konteyner içerisine yerleştirilen bir mikroservis, daha fazla hesaplama kaynağına ihtiyaç duyulduğunda konteynerlerin çoğaltılması ile ölçeklendirilmektedir. Çoğu mikroservis yüksek kaynak taleplerine sahip olmakla birlikte bu servislerin belirli performans standartlarını yerine getirmesi gerekmektedir. Bu durum da mevcut kaynakları yönetememe, maliyet artışı, hizmette aksaklık ve enerjinin boşa harcanması gibi problemlerin oluşmasına neden olmaktadır. Bu yüksek servis gereksinimleri en verimli yerleşimi bulma ihtiyacını zorunlu kılarak her servisin kaynakları israf etmeden yeterli kaynağının olması sağlanmalıdır. Bu nedenle bulut kullanıcıları kaynak kullanımını en üst düzeye çıkarmak için

kaynaklarını planlamak istemektedir. Sanallaştırılmış bulut ortamlarında kaynakların nasıl yönetileceği konusunda literatürde bir çok öneri sunulmuştur [11–13]. Bunlardan birisi kaynak planlamayı sağlamak için, sanal makine (Virtual Machine-VM) yerleştirme problemi (VM Placement problem-VMP) olarak adlandırılan fiziksel makineler üzerinde VM'lerin verimli yerleştirilmesi için önerilen planlama (scheduling) algoritmalarıdır [14]. VMP hakkında literatürde bir çok araştırma yapılmasına rağmen hem mikroservislerin hem de konteynerlerin planlaması yeni bir araştırma konusudur. Pratikte mikroservislerin dağıtımı ve konfigürasyon işlemleri bir uzman tarafından manuel olarak yapılmaktadır. Kubernetes [15], Docker Swarm [16] ve Apache Mesos [17] gibi otomatik dağıtım yapan konteyner teknolojileri kullandığında da bir uzman tarafından manuel olarak hazırlanan bir konfigürasyon dosyasına ihtiyaç duyulmaktadır. Bu süreçte servis dağıtımı verilen konfigürasyona göre yapılamazsa sistemin büyüklüğüne bağlı olarak uzman tarafından yeniden konfigürasyon dosyasının hazırlanması ve sürecin tekrarlanması uzun zaman almaktadır ve en verimli dağıtımın manuel olarak yapılması sistemin büyüklüğüne göre giderek zorlaşmaktadır. Bu problem ile başa çıkmak için otomatik verimli dağıtım alternatifleri üreten sistematik ve biçimsel yaklaşımlara ihtiyaç duyulmaktadır.

Bu tez çalışmasında mikroservis tabanlı mimarilerin tasarım aşamasında Kapasite Kısıtlı Görev Atama Problemi'ni (Capacitated Task Assignment Problem-CTAP) çözmeye yönelik algoritmalar kullanılarak verimli dağıtım alternatifleri üreten sistematik bir yaklaşım önerilmektedir. Yaklaşım farklı hafıza kapasiteleri ve hesaplama güçlerine sahip fiziksel kaynaklar üzerinde tanımlı verimli dağıtım alternatifleri türetmektedir. Önerilen yaklaşımda öncelikle benzetim ortamı bileşenleri ve servislerin dağıtılacağı fiziksel kaynakların yer aldığı altyapı modeli tasarlanmaktadır. Mikroservis Veri Değişim Modeli ve Mikroservis Altyapı Modeli'ni kullanan Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli de tasarlandıktan sonra mikroservisler için dağıtım alternatifleri algoritmik olarak türetilebilmektedir. Ayrıca algoritmik olarak türetilen dağıtım alternatiflerini değerlendirmek adına önerilen yaklaşımı destekleyen bir araç ailesi geliştirilerek çeşitli durum çalışmaları [5, 18] üzerinden

yaklaşım test edilmektedir. Yaklaşımı destekleyen araç, kullanıcıya algoritmik olarak hesaplanan iletişim ve çalışma maliyetlerini detaylı bir analiz ve değerlendirme raporu ile karşılaştırmalı bir şekilde kullanıcıya sunarak yazılım tasarım aşamasında bir veya daha fazla verimli dağıtım senaryosu belirlenmesine olanak tanımaktadır.

Bu tez çalışması kapsamında önerilen yaklaşım ve araç ailesinin katkıları aşağıda maddeler halinde listelenmektedir:

- Önerilen yaklaşım, sistem tamamen geliştirilmeden ve devreye alınmadan önce erken tasarım aşamasında farklı toplam iletişim ve yürütme maliyetleri ile geliştirme ekibine birden fazla dağıtım alternatifi sunmaktadır.

- Geliştirme yaşam döngüsünün sonraki aşamalarında yapılan değişikliklerin maliyeti katlanarak arttığından, yaklaşım; aşırı servis etkileşimleri, yüksek bellek gereksinimleri gibi tasarım sorunlarının erken tasarım aşamasında tespit edilmesine yardımcı olmaktadır.

- Dağıtım işlemi için servisler arası iletişim maliyetleri, servislerin sunucular üzerindeki çalışma maliyetleri, servislerin ve sunucuların hafıza kapasitesi gibi kısıtlar dikkate alınmaktadır. Bu kısıtlamaları iyi bilinen Kapasite Kısıtlı Görev Atama Problemi'ne (Capacitated Task Assignment Problem-CTAP) [19] uyarladıktan sonra, yaklaşım tasarım aşamasında alternatif dağıtım modelleri oluşturmak için farklı algoritmaların uygulanmasına izin vermektedir.

- Oluşturulan dağıtım modelinin mikroservis tasarım analizi ve alternatif dağıtım yaklaşımlarının karşılaştırılması dahil olmak üzere yaklaşımın her adımının gerçekleştirilmesini destekleyen bir araç seti tasarlanmış ve geliştirilmiştir.

- Bu araç seti ile otomatik bir şekilde verimli dağıtım alternatifleri türetmek, binlerce mikroservisten oluşan bir sistem için yapılan manuel dağıtıma göre çok daha az zaman almaktadır.

2. GENEL BİLGİLER

Bu bölümde önerilen yaklaşımı ve araç ailesini destekleyen ve anlaşılır hale getiren bilgiler yer almaktadır. Bölüm 2.1.'de mikroservislerin yapısı ve diğer yaygın mimariler ile karşılaştırılması yer almakta, Bölüm 2.2.'de literatürde yer alan mevcut mikroservis dağıtım alternatifleri açıklanmaktadır. Bölüm 2.3.'de mikroservis tabanlı uygulamaların verim dağıtım problemi açıklanarak Bölüm 2.4.'de önerilen yaklaşımı gerçekleştirmek için ele alınan Kapasite Kısıtlı Görev Atama Problemi (Capacitated Task Assigment Problem-CTAP) kısaca açıklanmaktadır. Bölüm 2.5.'de ve Bölüm 2.6.'da mikroservisler için literatürde yer alan mevcut iletişim desenleri ve mikroservislerin dağıtımı ve iletişimi esnasındaki zorluklar sırası ile anlatılmaktadır. Bölüm 2.7.'de bu zorluklara yönelik araştırma yönergeleri ve anahtar bulgulardan bahsedilmektedir. Bölüm 2.8.'da önerilen yaklaşım ve araç için uygulanan model güdümlü geliştirmeden bahsedilmektedir. Model güdümlü geliştirme için popüler araçlardan Eclipse Modelleme Aracı (Eclipse Modeling Tool) Bölüm 2.9.'da detaylı olarak anlatıldıktan sonra bu bölüm sonlanmaktadır.

2.1. Mikroservis Konsepti

Kurumsal yazılım uygulamaları, bir çok iş gereksinimini karşılamak için yüzlerce fonksiyonelliğe sahiptir ve bu fonksiyonellikler genellikle tek parçalı (monolithic) bir uygulama kapsamında tasarlanmaktadır. Sistemler ve projeler geliştikçe, uygulama güncelleme, ölçekleme ve takımlar halinde çalışma ihtiyacı artmakta ve yek pare bir uygulamada sadece talebi fazla olan fonksiyonu ölçekleme, tek bir yerde güncelleme gibi işlemler imkansız hale gelmektedir [20]. Bu gibi durumlarda en ufak bir değişiklikte tüm uygulamanın değişmesi söz konusudur ve tüm uygulama için sorun giderme, sürüm yükseltme, sunucuya yerleştirme gibi işlemler büyük maliyet gerektirmektedir. Bu problemler ile başa çıkmak adına "servis" kavramı ortaya çıkarılarak Servis-Odaklı Mimari (Service-Oriented Architecture-SOA) kullanılmaya başlanmıştır [21]. Servis sağlayıcı ve servis tüketici olarak iki ana role sahip SOA'nın konseptinde bir uygulama, modüllerinin sorunsuz bir şekilde entegre edilebileceği ve kolayca

yeniden kullanılabileceği şekilde tasarlanıp inşa edilebilecek şekilde olmalıdır. SOA sayesinde fonksiyonlliklere ayrılan servislerin başka uygulamalarda yeniden kullanımı, daha kolay yönetilebilirlik, daha yüksek güvenilirlik ve geliştirme sürecinde farklı takımlar halinde paralel geliştirme olanağı bulunmaktadır [21]. Fakat belirli fonksiyonelliğe sahip servislerin yönetimi hala karmaşıktır ve bu servislerin ayrışımı ekstra yük getirmektedir.

SOA'nın mimari yapısından esinlenen mikroservis mimarilerinde servisler küçük, otonom ve birbirlerine gevşek bağlı bir biçimde daha granül bir yapıya sahiptir. Mikroservis mimarisinin temeli, kendi süreçlerinde çalışan, bağımsız olarak geliştirilebilen ve dağıtılabilen birçok servisin birleşiminden oluşan tek bir uygulamaya dayanmaktadır [22]. Servislerin birbirlerinden bağımsız yapıda olması servisler arası etkileşimi, dağıtımı ve ölçeklendirmeyi kolaylaştırmaktadır.

Mikroservis mimarileri (Microservice Architectures-MSA) ve SOA karşılaştırıldığında her ikisi de ana bileşen olarak servisler üzerinden çalışmasına rağmen servis karakteristikleri açısından büyük farklılıklar göstermektedir. SOA "mümkün olduğunca paylaş" yaklaşımını benimserken, mikroservisler "mümkün olduğunca az paylaş" yaklaşımını benimsemektedir [23]. SOA iş işlevselliğinin yeniden kullanımına önem verirken mikroservis mimarileri "sınırlı bağlam (bounded context)" kavramı üzerine yoğunlaşmaktadır. İletişim desenleri incelendiğinde ise SOA iletişim altyapısı olarak ESB (Enterprise Service Bus) kullanırken, mikroservisler genellikle basit mesajlaşma sistemini kullanmaktadır. Tüm servisleri birbirine bağlayan ESB sisteminde tek nokta hatası (single point of failure) olasılığı çok yüksektir [24]. ESB'nin kapanması durumunda müşteriler ve servisler arasında iletişim kurulamamakta, tüm sistem çalışamaz hale gelmektedir. Diğer bir dezavantaj ise fazladan dolaylı aktarım seviyesi, müşteri-servis iletişiminin performansının düşmesine neden olmaktadır [24]. Öte yandan mikroservisler hata toleransı açısından daha kullanışlıdır. Bir mikroserviste oluşan herhangi bir problem sadece o servisi etkilemekte ve diğer tüm mikroservisler gelen istekleri yerine getirmeye devam edebilmektedir. Kullanılan protokoller incelendiğinde ise SOA heterojen mesajlaşma protokollerini desteklerken, mikroservisler genellikle HTTP/REST gibi daha hafif (lightweight) protokolleri desteklemektedir. Yapılan uygulamanın amacına göre hangi mimarinin daha etkili çalışabileceği değişmektedir. Her iki yaklaşım da

bazı avantajlara sahip olduğu için yaklaşımlardan herhangi biri uygulama ortamının büyüklüğü ve çeşitliliğine göre tercih edilebilmektedir. ESB aracılığı ile heterojen uygulamalar ve mesajlaşma protokollerinin entegrasyonunu destekleyen SOA daha geniş bir ortam çeşitliliğine katkıda bulunurken, mikroservisler web ve mobil uygulamalar gibi daha küçük ve daha bağımsız ortamlarda farklı ekipler tarafından geliştirilmek için tercih edilmektedir [25].

Ayrıca SOA ve MSA yaklaşımları gerektiği zaman farklı seviyelerde birlikte kullanılabilir. Her sorunun çözülebildiği bir konsept olmayan MSA, dağıtık sistemlerden kaynaklanan karmaşıklığı da beraberinde getirmektedir. Farklı servislerin birbirleri ile iletişiminin sağlanabilmesi geleneksel veritabanı işlemleri ile yapılamamakta, servislerin testi zorlaşmakta ve kaynakların özelliklerine göre servislerin performanslı bir şekilde dağıtımı bir problem haline gelmektedir. Fakat uzun vadede zorlukların çözümü düşünüldüğülde mikroservis mimarilerinin daha tercih edilebilir olduğu görülmektedir. Çünkü, MSA'daki işlevselliklerin ayrıştırılması, geliştirici çevikliğini, bağımsız olarak ölçeklenebilirliği ve esnekliği basitleştirmektedir [25]. Ayrıca, MSA'nın dağıtım süreci SOA'dan daha kolaydır, çünkü servisler daha küçük ve gevşek bağlı olarak tasarlandığından, birbirlerinden bağımsız olarak konuşlandırılabilirler. Bu nedenle, işletmeler genellikle kod temelinin karmaşıklığı, ölçeklendirme zorluğu, tüm uygulamanın dağıtım zorluğu ve esneklik sorunları gibi tek parçalı uygulamaların ve SOA'nın bazı dezavantajları nedeniyle mikroservislere yönelme eğilimindedirler [20].

2.2. Mevcut Mikroservis Dağıtım Modelleri

Mikroservislerin bir sanal makine veya fiziksel sunucular gibi hesaplama kaynaklarını temsil eden düğümlere dağıtımı için mevcut olan birkaç dağıtım yöntemi bulunmaktadır. Bu yöntemler (1) "Her sunucu modeli için çoklu servis örneği", (2) "Her sunucu modeli için bir servis örneği" ve (3) "Sunucusuz dağıtım" olmak üzere üç gruba ayrılmaktadır [26].

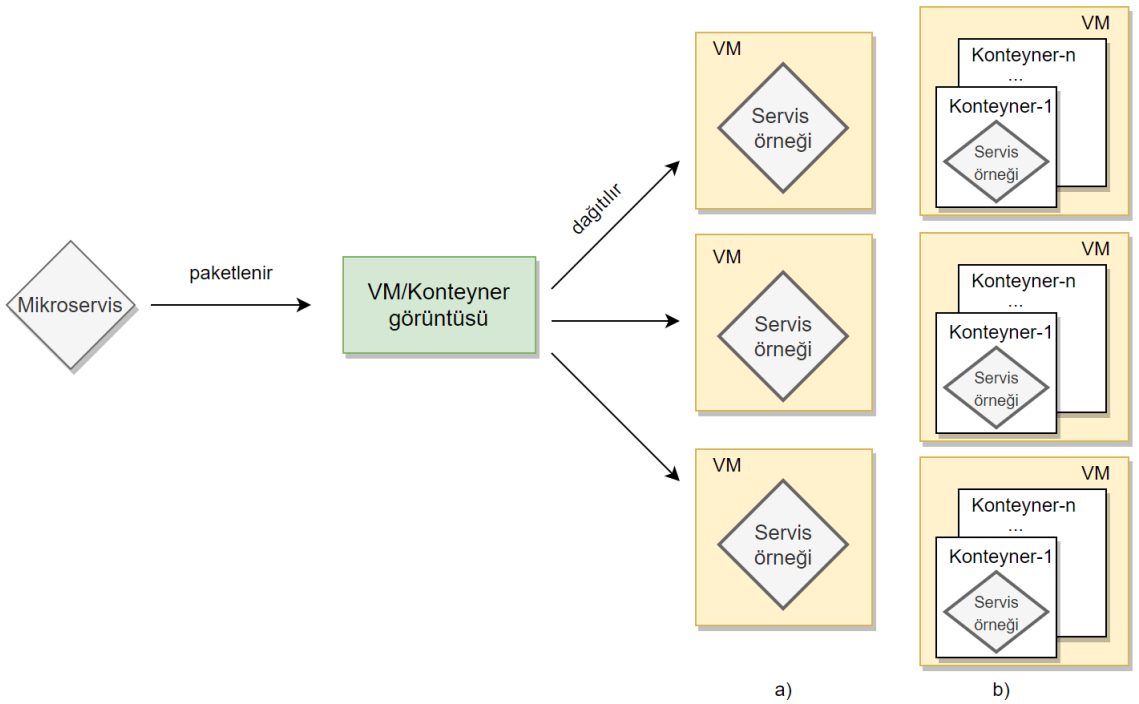
Her sunucu modeli için çoklu servis örneğinde, bir servis için bir veya daha fazla fiziksel veya sanal sunucular hazırlanmakta ve bu sunucuların her biri üzerinde birden çok servis örneği çalıştırılabilmektedir. Her servis örneği, bir ya da daha fazla sunucu üzerinde bilinen bir portta çalışmaktadır. Hem avantajlara hem de dezavantajlara sahip çoklu servis modelinde,

örnekler aynı sunucuyu ve o sunucunun işletim sistemini paylaştıkları için bu model kaynak kullanımı açısından etkilidir. Diğer bir avantajı ise bu model ile bir servis örneğinin dağıtımı hızlı bir şekilde yapılabilmektedir. Bu modelin en önemli dezavantajı ise her servis örneğinin ayrı bir görevi olmadığı sürece servis örneklerinin yalıtımının çok az ya da hiç olmamasıdır. Bu yalıtım eksikliğinden dolayı kötü niyetli bir servis, aynı görev içerisinde çalışan diğer servisleri kolayca istismar edebilmektedir. Her servis örneğinin kaynak kullanımı doğru bir şekilde izlenmesine rağmen her bir örneğin kullandığı kaynaklar sınırlanamamaktadır. Sunucunun tüm belleğini veya CPU'sunu tüketmek için servis örneğinin kötüye kullanılabilir. Bu yaklaşım ile ilgili bir başka problem de bir servisi dağıtan operasyon ekibinin bunu nasıl yapacağına dair belirli detayları bilmesi gerektiğidir. Servisler çeşitli dillerde ve çerçevelerde yazılabilir, bu yüzden geliştirme ekibinin operasyon birimleri ile paylaşması gereken birçok ayrıntı bulunmaktadır. Bu karmaşıklık, dağıtım esnasındaki hata riskini artırmaktadır.

Çoklu servislerin aynı sunucuda çalıştığı dağıtım modelinin problemlerini önlemek adına "her sunucu modeli için bir servis örneği" önerilmiştir. Bu dağıtım modelinde çoklu servislerin izolasyon problemi yer almaktadır. Bu problem her bir servis örneği kendi sunucusunda ayrı ayrı çalıştırılarak çözülmektedir. Bu model, "her sanal makine (VM) başına servis örneği" ve "her konteyner başına servis örneği" olmak üzere iki farklı şekilde özelleşmektedir. Her sanal makine başına servis örneği modelinde Amazon EC2 AMI [27] gibi VM görüntüsü olarak her bir servis paketlenmektedir ve her bir servis örneği o VM görüntüsünü kullanarak başlatılan bir sanal makinedir. Bu dağıtım modeli kendi video işleme servisini dağıtmak için Netflix tarafından kullanılan birincil yaklaşımdır. Çoklu servis örneğinde olduğu gibi bu yaklaşımda da çeşitli avantajlar ve dezavantajlar bulunmaktadır. VM'ler sayesinde her bir servis örneği diğer servislerden tamamen izole bir şekilde çalışmaktadır. Sabit miktarda CPU ve belleğe sahiptir ve diğer servislerden kaynak çalma durumu yoktur. Model, bu özelliği ile çoklu servis yaklaşımının izolasyon ve güvenlik problemlerine çözüm olmuştur. Mikroservislerin VM'ler kullanılarak dağıtılmasının bir diğer avantajı da bulut altyapısının kullanılabilmesidir. AWS [27] gibi bulutlar, yük dengeleme ve otomatik ölçeklendirme gibi özellikler sunmaktadır. Servislerin bir VM olarak dağıtılmasının bir başka faydası, sanal makinenin servisin uygulama teknolojisini sarmalamasıdır. Bir servis, VM tarafından paketlendiği zaman,

o servis kara kutu haline gelmekte ve bu servise dışardan müdahale oldukça zorlaşmaktadır. VM'in yönetim yazılım kütüphanesi, servis dağıtımında kullanılmakta ve dağıtım basit ve güvenilir bir hale gelmektedir. Modelin bir dezavantajı, daha az verimli kaynak kullanımıdır. Her servis olgusu, işletim sistemi dahil olmak üzere tüm VM'in ek yüküne sahiptir ve genel bir IaaS'de VM'ler sabit boyutlarda gelirler ve VM'in yeterince kullanılamaması mümkündür. Ayrıca genel bir IaaS, sanal makinelerin meşgul ya da boşta olduğuna bakılmaksızın sanal makineler için ücret almaktadır. Bu da dağıtım maliyetini artırmaktadır. Yaklaşımın bir diğer dezavantajı, VM görüntülerinin boyutları nedeniyle oluşturulmaları yavaş olduğundan bir servisin yeni versiyonunu dağıtmak genellikle yavaştır. Ayrıca, VM'lerin tipik olarak, büyüklükleri nedeniyle tekrar başlatılma süreçleri yavaş ilerlemektedir. VM'lerin bahsedilen faydalarını kullanmak ile birlikte daha hafif alternatif yollardan biri olan "Her konteyner başına servis örneği" ortaya çıkmıştır. Bu dağıtım modelinde her bir servis örneği kendi konteynerinde çalışmaktadır. Konteynerler, işletim sistemi düzeyinde çalışan bir sanallaştırma mekanizmasıdır. Bir konteyner, sanal alanda çalışan bir veya birden fazla işlemden oluşmaktadır. Konteynerlerin kendilerine ait port ad alanı ve kök dosya sistemleri vardır. Konteynerlerin belleği ve CPU kaynakları sınırlandırılabilmektedir. Bazı konteyner uygulamaları G/Ç oranı sınırlandırmasına da sahiptir. Bu servis modelini kullanmak için servisin bir konteyner görüntüsü olarak paketlenmesi gerekmektedir. Bir konteyner görüntüsü, servisi çalıştırmak için gereken uygulamalar ve kütüphanelerden oluşan bir dosya sistemi görüntüsüdür. Servis bir konteyner görüntüsü olarak paketlendikten sonra bir ya da daha fazla konteyner başlatılmaktadır. Genellikle her fiziksel ya da sanal sunucu üzerinde birden fazla konteyner çalıştırılmaktadır. Aynı zamanda konteynerleri yönetmek için Kubernates ya da Marathon gibi bir küme yöneticisi de kullanılabilmektedir. Bir küme yöneticisi, sunuculara bir kaynak havuzu gibi davranmaktadır. Her bir sunucuda bulunan kaynaklar ve konteynerin gerektirdiği kaynaklara bağlı olarak her bir konteynerin nereye yerleştirilmesi gerektiğine karar vermektedir.

Konteynerler de VM'ler gibi benzer avantajlara sahiptir. Konteyner teknolojisi ile servis örnekleri birbirlerinden izole edilerek her konteyner tarafından tüketilen kaynakların kolayca



Şekil 2.1. a) Her VM için bir servis örneği b) Her konteyner için bir servis örneği

izlenmesi sağlanmaktadır. Ayrıca VM'ler gibi konteynerler de servisleri uygulamak için kullanılan teknolojiyi sarmalamakta ve güvenli bir yapı haline getirmektedir. Konteyner yönetim API'si, servisleri yönetmek için API görevi de görmektedir [26]. Fakat VM'lerin aksine konteynerler esnek teknolojilerdir ve konteyner görüntülerini oluşturmak ve başlatmak genellikle çok hızlıdır. Konteyner teknolojisi kullanımının dezavantajı, konteyner görüntülerinin yönetiminden geliştiricinin sorumlu olmasıdır. Google Container Engine veya Amazon EC2 Container Service (ECS) gibi bir hizmet kullanılmıyorsa, konteyner altyapısının ve muhtemelen üzerinde çalıştığı VM altyapısının yönetilmesi gerekmektedir. Ayrıca, konteynerler genellikle VM'lere göre ayarlanmış fiyatlandırmaya sahip bir altyapı üzerine yerleştirilmektedir. Bu nedenle, daha önce açıklandığı gibi, ani yüklerin üstesinden gelebilmek için VM'lerin ekstra maliyetine maruz kalmak muhtemel bir durum haline gelmektedir. "Her sunucu modeli için bir servis örneği" modellerinin yapısı Şekil 2.1.'de gösterilmektedir.

Tüm bu mikroservis dağıtım stratejilerinin yanı sıra yeni bir yöntem olan sunucusuz dağıtım modeli bulunmaktadır. Bu model, konteyner veya sanal makinelerde servis dağıtımı

arasında seçim yapmak zorunda kalmayı engelleyen bir yaklaşımdır. Sunucusuz dağıtım teknolojisinin bir örneği olan AWS Lambda'da [28] Java, Node.js ve Python servislerini desteklemektedir. Mikroservisler AWS Lambda'ya [28] zip dosyası olarak yüklenerek dağıtım gerçekleştirilmektedir. Aynı zamanda bir isteği işlemek için çağrılan fonksiyonun adını belirten üst veriler de tedarik edilebilmektedir. AWS Lambda [28] istekleri işlemek için yeterli miktarda mikroservisi otomatik olarak çalıştırmaktadır. Alınan süreye ve tüketilen belleğe bağlı olarak her istek için kullanıcı faturalandırılmaktadır. Mikroservislerin dağıtımında uygun bir yöntem olan sunucusuz dağıtım modeli, geliştiriciyi bilgi teknolojisi altyapısından sorumlu olmadan uygulama geliştirmeye odaklamaktadır. Aynı zamanda istek tabanlı fiyatlandırma, yalnızca servislerin gerçekleştirdiği iş için ödeme yapıldığı anlamına gelmektedir. Sunucusuz dağıtımın tüm bu avantajları ile birlikte bazı sınırlamaları da bulunmaktadır. AWS Lambda [28] işlevleri, üçüncü taraf bir mesaj kırıcısından (message broker) gelen mesajları tüketen bir servis gibi uzun süredir devam eden servisleri dağıtmak için kullanılamamakta ve isteklerin 300 saniye içinde tamamlanması gerekmektedir. Ayrıca teoride AWS Lambda, her talep için ayrı bir olgu çalıştırabildiği için servisler durumsuz olmalıdır ve sadece desteklenen dillerden birinde yazılmalıdır. Servisler aynı zamanda hızlı bir şekilde başlamalıdırlar; aksi halde zaman aşımına uğramakta ve sonlandırılabilmektedirler.

2.3. Mikroservis Tabanlı Uygulamaların Verimli Dağıtım Problemi

Mikroservislerde en uygun dağıtım problemi, verilen bir mikroservis sisteminde, mevcut düğüm kümelerine dağıtılacak yeni bir hedef mikroservisin düğümler üzerindeki çalışma maliyeti, bir mikroservisin sunucu üzerine yerleşmesi için gerekli hafıza miktarı, mikroservislerin birbirleri ile veri alışverişi sırasında elde edilen iletişim maliyeti gibi maliyetlerin minimuma indirgenerek sistemdeki toplam maliyetin en iyilenmesi şeklinde tanımlanabilmektedir [29]. Her bir düğümün en fazla polinomsal miktarda mikroservisi barındırabileceği varsayımı altında dağıtım problemi, polinomsal zamanda çözülebilen zor bir problem olarak NP-tam şeklinde tanımlanmaktadır. Toplam maliyeti en aza indirgeyen bir sistemin dağıtılması sorunu da bir NP-optimizasyon problemidir. [29].

Bir kümede mikroservis tabanlı uygulamanın dağıtımı, mühendisler tarafından tanımlanan gerekli kaynaklar ve ana bilgisayarlarda mevcut olan kaynaklar dikkate alınarak yapılmaktadır. Dağıtımı yapılandırmak için uygulamayı geliştiren mühendisler mikroservislerin CPU ve hafıza gibi ihtiyaç duydukları minimum ve maksimum kaynakları ayarlayabilmektedir. Fakat bu değerleri doğru bir şekilde aktarmak için standartlaştırılmış bir kural bulunmamaktadır. Mühendisler genellikle bu değerleri mikroservislerin önceki çalıştırılmalarına veya öznel olan kendi deneyimlerine göre belirlemektedir. Bu nedenle bir mikroservisin çalışma zamanında performanslı çalışması için hangi kaynaklara ihtiyaç duyabileceğini belirlemek zor bir süreçtir.

Kubernetes [15], Docker Swarm [16], Apache Mesos [30] gibi yönetim araçları, kaynak kullanımının en üst sınırının kullanıcı tarafından belirlenmesine izin vermesine rağmen, mühendislerin bu sınırı net bir şekilde belirleyeceklerinin bir garantisi yoktur. Sınırlar doğru belirlense bile seçilen değerlerin mikroservis tabanlı uygulamalar için sistemin en iyi maliyet ile sonuç vereceğinin de garantisi bulunmamaktadır. Bu durumda yeniden bir konfigürasyon dosyası hazırlanarak dağıtımın tekrar yapılması gerekmektedir. Uygun dağıtım alternatifinin bulunmadığı durumlarda bu ayarların yeniden yapılması algoritmik bir yaklaşıma göre oldukça yavaştır.

Eğer sadece minimum gerekli kaynak miktarı ayarlanırsa bir çok mikroservisin bir arada tek bir ana bilgisayara yerleştirilmesi olasıdır. Aynı ana bilgisayara yerleştirilmiş mikroservisler, her ne kadar aynı düğümde oldukları için iletişim gecikmesinin sıfıra indigenmesini sağlasa da sunucular arasında yük dengelemeyi önlemektedir. Ayrıca bu durumda toplamda sunucu üzerinde mevcut olandan daha fazla kaynak talep edilebilmektedirler. Her ne kadar yönetim araçları sayesinde tek bir düğüme dağıtma işlemi önlense de bu araçlar mikroservislerin çalışma zamanındaki ihtiyaçlarından haberdar değillerdir. Dağıtım zamanında küme sağlayıcı, mikroservis tabanlı uygulamanın performansını tehlikeye atmadan sunucular arasındaki yük dağılımını (kaynak kullanımı) dengelemeye çalışmaktadır. Fakat mühendislerin mikroservis kaynak ihtiyacını belirleme konusunda standartlaşma eksikliği, bu servislerin yerleştirilmelerini zorlaştırmaktadır. Spread, bin-pack, labeled ve random gibi birçok strateji kullanarak yerleştirme yapan yönetim araçları, hangi stratejiyi kullanırsa kullansınlar mikroservislerin

yerleşimini yönlendirmek veya geliştirmek için geçmiş verileri kullanmamaktadır [31]. Mevcut araçlar, mikroservisi yerleştirmek için anlık kaynak kullanımını göz önünde bulunduran ana bilgisayarları seçmekte ve nadiren en uygun dağıtımı bulmaya çalışmaktadır.

Yönetim araçlarının bu problemlerinden dolayı bu tez kapsamında mikroservislerin mesaj değiş tokuşu sırasındaki iletişim maliyetleri, sunucular üzerindeki çalışma maliyetleri, barındırılması için gereken hafıza miktarları, aynı zamanda sunucuların da mikroservisleri barındırmaları için gerekli hafıza ve CPU miktarları göz önüne alınarak algoritmik bir yaklaşımla minimum toplam maliyeti elde etmek amaçlanmaktadır. Bu şekilde en uygun dağıtım yöntemi bulunamadığı durumlarda sistemin algoritmik çözüm ile geçmiş verileri kullanarak uygun bir alternatif bulması sağlanmaktadır. Böylece yönetimsel araçlara göre daha hızlı ve en iyiye daha yakın çözümler üretilmesi amaçlanmaktadır.

2.4. Kapasite Kısıtlı Görev Atama Problemi (Capacitated Task Assignment Problem-CTAP)

Mikroservislerin en uygun düğüme yerleştirme işlemi yapılırken servislerin depolanması için gerekli hafıza, servislerin düğümler üzerindeki çalışma maliyeti, düğümlerin hafıza kapasiteleri gibi bir çok parametreye ihtiyaç duyulmaktadır. Bu kısıtların bir arada bulunduğu yerleştirme problemi literatürde "Capacitated Task Assignment Problem (CTAP)" ile eşleşmektedir [19]. Mikroservislerin verimli dağıtımı için kullanılan bu problem kısaca aşağıdaki gibi tanımlanmaktadır:

CTAP, m adet düğüme n adet işin minimum çalışma ve iletişim maliyeti ile atanması problemidir. NP-zor olarak bilinen bu problem Görev Atama Probleminin (Task Assigment Problem-TAP) bir uzantısıdır. CTAP'de TAP'den farklı olarak kaynakların hafıza kapasiteleri kısıtlıdır ve görevler arası iletişim maliyetleri atama esnasında göz önünde bulundurulmaktadır. Bu problem, dağıtık bilgisayar sistemlerinde bir model olarak kullanılmakla birlikte iş zamanlama, araç yönlendirme, en kısa yol bulma, telekomünikasyon ağ tasarımı, kargo yükleme gibi problemler için de kullanılmaktadır. Mikroservis mimarilerinin dağıtımında kullanılmak için önerilen araç için model alınan CTAP'de amaç, mikroservis örnekleri arasındaki

iletişim maliyetini ve servislerin düğümlere dağıtımı sonucunda oluşan çalıştırma maliyetini minimuma indirgemektedir. Minimuma indirgeme işlemi yapılırken ilgili düğüme dağıtılan servislerin toplam hafıza kapasitesi ve işlemci gücünün, düğümün hafıza kapasitesini ve işlemci gücünü aşmaması gerekmektedir. Bir iş atama problemi olan CTAP yönteminin genel tanımı yapılırsa, m adet iş olan bir ortamda i işi m_i birim belleğe ihtiyaç duyar. Ortamda n adet birbirine denk olmayan işlemci vardır, p işlemcisinin toplam M_p birim belleği ve C_p birim işlem kapasitesi bulunmaktadır. i işini p işlemcisi üzerinde çalıştırmanın maliyeti x_{ip}'dir. C_{ij}, i ve j işleri arasındaki toplam iletişim maliyetini göstermektedir. İletişim maliyetleri hesaplanırken iletilen veri boyutuna ek olarak işler arası iletişimin sıklığı dikkate alınmalıdır. i ve j işleri arasındaki yüksek iletişim sıklığı daha yüksek iletişim maliyeti C_{ij}'ye sebep olacaktır. Eniyileme problemindeki amaç, işleri işlemcilere bellek ve işlem gücü kısıtlarını aşmadan asgari iletişim ve işletim maliyetiyle yerleştirmektir. Problemin karar değişkeni: a_{ip}= 1, eğer i işi p işlemcisine atanmışsa, diğer durumda 0 olmaktadır. Bu tanım doğrultusunda amaç matematiksel olarak ikili karar değişkenleri olan bir eniyileme problemi (optimization problem with binary decision variables) olarak Şekil 2.2.'deki gibi formulize edilmektedir.

(**M**) Enküçükle $\sum_{i=1}^{m}\sum_{p=1}^{n} a_{ip}x_{ip} + \sum_{i=1}^{m}\sum_{j=1}^{m}\sum_{p=1}^{n} a_{ip}(1-a_{jp})c_{ij}$

Şu kısıtlar dahilinde:

$$\sum_{p=1}^{n} a_{ip} = 1, \qquad \forall i = 1,\ldots..,m$$

$$\sum_{i=1}^{m} m_i a_{ip} \le M_p, \quad \forall p = 1,\ldots..,n$$

$$\sum_{i=1}^{m} x_{ip} a_{ip} \le C_p, \quad \forall p = 1,\ldots..,n$$

$$a_{ip} = \{0,1\}, \qquad \forall i = 1,\ldots..,m, \ \forall p = 1,\ldots..,n$$

Şekil 2.2. CTAP tanımı

2.5. Mikroservisler için Mevcut İletişim Desenleri

Monolitik bir mimaride iletişim altyapısının kurulması için, veritabanı tabloları arasındaki ilişkilerin belirlenmesi ve bu veritabanının bir nesne modeline eşlenmesi yeterlidir. Ancak mikroservis mimarilerinde her servis ayrı veritabanlarında depolandığı için ihtiyaç halinde bu servislerin uygun iletişim kalıplarını kullanarak birbirleriyle etkileşimde bulunmaları gerekmektedir. Bu çalışma kapsamında mikroservis mimarileri için modellenecek olan iletişim protokollerini belirlemek için literatürde yer alan mevcut iletişim desenleri aşağıdaki gibi listelenmektedir.

Eşzamanlı (senkron) iletişim: HTTP tabanlı REST ve istek/cevap tabanlı Apache Thrift gibi senkron iletişim mekanizmalarında istemci ihtiyaç duyduğu veriyi elde etmek için ilgili servisten cevap gelene kadar meşgul durumunda kalmaktadır. Bir başka senkron iletişim türü olan istek/yanıt adı verilen bire bir etkileşim (one-to-one interaction) türünde [32] istemci, bir istek gönderdikten sonra yanıtı beklemektedir. Bir yanıt belirli bir zaman aşımı süresince istemciye bilgi vermez ise, istemci hata hakkında bir bildirim almaktadır.

Yayınla/Abone ol (publish/subscribe) iletişimi: Olay güdümlü mimariye dayalı bir iletişim deseni olan bu iletişim türünde tüm olaylar bir konu üzerinde yayınlanır ve yayıncı servise abone olan tüm servisler yayınlanan veriyi kullanabilmektedir.

HTTP ve mesaj kuyruğunun birleşimi: Birbirleri ile veri iletişiminde bulunan iki mikroservis, hem eşzamanlı (senkron) hem de eşzamansız (asenkron) iletişim desenleri ile hibrit bir model kullanabilmektedir. İlk olarak veri alışverişi senkron iletişim ile gerçekleştirilmektedir. Servislerden biri web servis işlevi sunuyorsa, başka bir servis bu servise HTTP üzerinden veri istekleri gönderebilmekte ve istenen verileri içeren bir yanıt alabilmektedir. Senkron iletişim modeli, istenen yanıtın anında iletilmesini sağlar, ancak diğer hizmetin bağlı bir hizmeti kullanabilmesini gerektirir. Talep edilen servis meşgul durumunda ise, talebi gönderen kullanıcı bilgilendirilmelidir. Bu durumda alternatif olarak asenkron bir iletişim modeli tercih edilmektedir. Gelen istek mesaj kuyruğuna yerleştirilerek ilgili veri mesaj kuyruğundan

kullanıcıya iletilmektedir. Çalışma [33], bu yaklaşımı yük dengelemeye bir çözüm olarak sunmaktadır.

Mesaj odaklı ara katman yazılımı (RabbitMQ, ActiveMQ, vb.) kullanarak iletişim: [34–36] çalışmaları, mesaj odaklı bir ara katman yazılımı kullanan mikroservis iletişim yöntemlerini önermektedir. Çalışma [34], mikroservis tabanlı bir web uygulamasında RabbitMQ ve RESTful API kullanarak bu katmanların iletişim performansını karşılaştırmaktadır. Mesaj odaklı bir ara katman yazılımında, bir API Gateway, bir kullanıcıdan gelen istekleri RESTful API kullanarak tüm servislerin adreslerini toplayarak ilgili servislere iletir. API Gateway isteği aldığında öncelikle yöntemi ve web uzantısını kontrol ederek RESTful API üzerinden isteği ilgili servise iletir. Talebi alan servis aynı şekilde metodu ve web uzantısını kontrol ederek bu talebi tanımlanan REST API ile işler. Bu iletişim türünde, servis örneklerinin birbirleriyle iletişim kurmasına ihtiyaç olmadan yalnızca yanıtların ağ geçidine iletilmesi yeterlidir. Ancak API Gateway, sistemde yalnızca bir ağ geçidi kullanıldığında tek bir hata noktasına neden olmaktadır. Bunu önlemek için API Gateway örneğini çoğaltma önerilmektedir. Diğer iletişim türünde API Gateway yerine RabbitMQ kullanılmaktadır. RabbitMQ, Değişim Fonksiyonu (Exchange Function) modülünü içermekte ve bu modül, belirtilen kuyruğa bir mesaj yayınlamaktadır. Değişim Fonksiyonu modülü tarafından bir mesaj alındıktan sonra, mesaj ilgili kuyruğa iletilmektedir. Kuyruğu takip eden bir servis modülü, sonraki mesajı alarak bu mesajı işlemektedir.

Eşzamansız (Asenkron) iletişim: Bu iletişim türünde, istemci talep ettiği veriyi mesajlaşma kuyruğu gibi asenkron bir medya aracılığıyla göndermekte ve yanıt beklerken diğer mikroservisler arasındaki iletişimi engellememektedir. Servisten gelen yanıt daha sonra istemciye iletilmektedir. RabbitMQ [37], Apache Kafka [38] ve Apache ActiveMQ [39] gibi mesaj kuyrukları, asenkron iletişimi uygulamak için endüstride yaygın olarak kullanılan popüler açık kaynaklı mesajlaşma sistemleri olsa da, literatürde istek/asenkron (request/asynchronous) ve yayınlama/asenkron (publish/asynchronous) olarak adlandırılan eşzamansız iletişim modellerine de ulaşılmaktadır [32]. İstek/asenkron modelde, istemci bir servise istek gönderir ve servis eşzamansız olarak yanıt verirken yanıtlar bir istek yayınlayan bir istemci

tarafından oluşturulmaktadır ve daha sonra yayınlamada ilgili servislerin yanıtı için asenkron bir şekilde belirli bir süre beklemektedir. Ayrıca [32, 40, 41] çalışmaları, mesaj kırıcılarını (message broker) kullanan asenkron iletişimi desenini tartışmaktadır. Bu iletişim ortamında, mesaj üretici servis bir kuyruğa mesajları iletmekte ve tüketiciler de bu kuyruktan asenkron olarak mesajları almaktadır. Bu iletişim türü, mesaj üreticileri ve mesaj tüketicileri arasındaki iletişimi bir ara mesaj kırıcısı aracılığıyla sağlamaktadır. Tüketiciler mesajları işleyene kadar mesaj kırıcı bu bilgileri saklamaktadır. Böylece mesaj kırıcıları üzerinden veri iletişiminde bulunan üretici ve tüketici mikroservisleri birbirinden tamamen habersizdir ve aralarında asenkron bir iletişim bulunabilmektedir.

Noktadan noktaya iletişim (Point-to-point communication): Noktadan noktaya iletişim deseninde, mesaj yönlendirme mantığı her bir uç noktada kalmakta ve her bir servis örneği, REST gibi API'leri kullanarak diğer servisler ile doğrudan iletişim kurabilmektedir. Ancak, bu iletişim türünün performansı, talep sayısı ve servislerin birbirleri ile iletişim karmaşıklığı arttıkça önemli ölçüde düşmektedir.

İkili protokoller kullanılarak iletişim (Communication using binary protocols): Bir mikroservis mimarisinde, birden fazla API ortaya çıkaran mikroservislerin sayısı hızla artabilmektedir. Bu durumda, her çağrı için iyi tanımlanmış bir arayüz sağlanarak ikili protokollerin kullanımı ile diğer iletişim protokollerinden daha verimli bir iletişim altyapısı kurulabilmektedir. Arayüz Tanımlama Dili (IDL), birçok farklı dilde geliştirilen servisler arasında sorunsuz iletişimin gerçekleşebilmesi için Thrift ve Protobuf gibi servis sahiplerini arayüz tanımlarını yayınlamaya mecbur kılmaktadır. Apache Thrift de alternatif bir ikili iletişim protokolü olarak kullanılabilmektedir [32, 42].

2.6. Mikroservislerin İletişim ve Dağıtımı ile İlişkili Zorluklar

2.6.1. Dağıtım Zorlukları

Mimari Karmaşıklık: Mikroservislerin bağımsız olarak geliştirilmesi ve dağıtılması; izleme, ölçeklendirme, sürekli geliştirme/teslimat (continuous development/delivery) ve kurtarma

(recovery) gibi farklı işlemlerin gerçekleşmesi sırasında karmaşıklığa neden olmaktadır. Bu nedenle, mikroservisleri manuel olarak yönetmek pratikte mümkün değildir [43].

Bağımsız olarak hataya uğrayan mikroservisler (Hata toleransı): Mikroservis tabanlı bir mimaride servislerin bağımsız olarak dağıtımı, sistemin tümünü etkileyecek hataları önlemek için servis hatalarının izolasyonunu da sağlamaktadır [44]. Bu durumda geliştirici, mikroservisleri dağıtırken hata toleransını dikkate almalıdır.

Dağıtım koordinasyonu: Bir servisin birden fazla servise ihtiyaç duyduğu birbirine bağlı bir grafik modelinde, servislerin dağıtımı esnasında sistemin sonsuz döngüye girmesi durumu söz konusu olabilmektedir. Bu sorunu çözmek için servisler Yönlendirilmiş Döngüsel Olmayan Grafik (Directed Acyclic Graph) üzerinde modellenmektedir. Aksi takdirde kullanıcı taşma (overflow) hataları ile karşılaşabilmektedir [45].

Dağıtık günlükler (Distributed logs): Bir uygulamada mikroservis örneklerinin dağıtımı, günlüklerin de dağıtık bir şekilde tutulmasına neden olmaktadır. Mikroservis tabanlı bir uygulamada bir sorun olduğunda, uygulamanın operatörleri sorunu çözmek için dağıtılan tüm günlükleri analiz etmekte zorlanabilmektedir [44].

Dağıtım maliyeti: Mikroservis tabanlı uygulamalarda her ekip, sorumlu olduğu servislerde güncelleme yaptığında servislerinin operasyonel ve uygulama maliyetlerinin farkında olmalıdır. Bir serviste yapılan bir değişiklik, güncellenen servis ile iletişimde olan diğer servisler üzerinde ek bir yüke neden olabilmekte ve bu servislerin ölçeklendirilme ihtiyacı ek maliyete sebep olmaktadır.

Konteyner görüntüsü zafiyeti: Mikroservisler büyük ölçekli sistemlerde işbirliği ilkesine sahip olduğundan, mikroservis tabanlı sistemlerde genellikle yeni servislerin devreye alınmasına ve mevcut servislerin güncellenmiş sürümlerine ihtiyaç duyulmaktadır. Mikroservisleri dağıtmak için endüstride sıklıkla kullanılan konteyner tabanlı teknolojiler (ör. Docker, Kubernetes, vb.), kullanıma hazır dağıtımları desteklemek için kara kutu yeniden kullanımını

benimsemektedir. Bu durumda konteynere dağıtılan servis içeriklerinin bozulmayacağı garantisi verilmemektedir. Böylece, kara kutu yeniden kullanım stili, servisin dağıtılacağı konteynerde taşınan bilgilerin gerçek olduğundan emin olmayı engellemektedir [46] .

Gerekli özel konfigürasyonlar: Bulut sunucularında (örneğin AWS) mikroservis mimarisinin dağıtımı yapılırken, ağ geçitleri ve servisler gibi birçok bağımsız uygulamanın dağıtımı da gerçekleşmektedir. Yeni bir ağ geçidi veya servis sürümü yayınlandığında, dış bağımlılıkları kırmak kolay olduğundan mikroservis mimarisi için servis versiyonlamasının sürekliliği önem arz etmekte ve bulut sunucusunun sunduğu uygulama ve servisler üzerinde spesifik konfigürasyonlara ihtiyaç duyulmaktadır [24].

Sürekli Entegrasyon/Sürekli Teslimat (Continuous Integration/Continuous Delivery-CI/CD): [47] referanslı çalışma, mikroservislerin kullanımının çeşitli avantajlar sağlamasının yanısıra, kısa sürüm döngüleri nedeniyle sürekli entegrasyon ve teslimat problemlerini beraberinde getirdiğini vurgulamaktadır. Ayrıca yazarlar, bir servisteki güncellemelerin veya yeni bir mikroservis ilavesinin mevcut bir uygulama ile daha az çaba ve daha az kesinti ile entegre edilmesi gerektiğini belirtmektedir. CD, bir serviste meydana gelen yeni özellik ekleme, hata onarma gibi değişikliklerin sürekli olarak hızlı ve güvenli bir şekilde yönetilme yeteneği iken CI, servislerdeki kod değişiklikleri genellikle tek bir ana dalda birleştirildiği için otomatik derleme ve test süreçleri, ana daldaki kodun her zaman üretim kalitesinde olmasını gerektirmektedir. Bu bağlamda sürekli entegrasyonun sağlanması mikroservislerin dağıtımı esnasında karşılaşılan önemli zorluklar arasında yer almaktadır.

2.6.2. İletişim Zorlukları

Servislerin keşfi (Discovery of services): Mikroservislerin iletişiminde en kritik konulardan biri sistem yer alan mevcut servislerin keşfedilmesidir. Her bir servis DNS'e kaydedilerek servis keşfi yapılabilmektedir ancak özellikle servis sayısının çok fazla olduğu durumlarda bu çözüm uygulanamaz hale gelmektedir [48]. Consul gibi servis düzenleme çözümleri, servis kaydı ve keşfi için sektörde yaygın olarak kullanılmaktadır [48].

Servis örneklerinin çoğaltılması: Servis örneklerinin çoğaltılması, çoğaltılan verilerin güncellenmesine yol açmaktadır. İsteklerin veya mesajların, istenen servisin tüm mevcut örneklerine adil bir şekilde dağıtılması gerektiğinden servis örneklerinin çoğaltılması bir zorluk haline gelmektedir. Veriler statik değilse ve servis sürekli güncel verilere ihtiyaç duyuyorsa, eşzamansız olarak çoğaltma yapmak mikroservisler için önemli iletişim zorlukları arasındadır [44].

Yük dengeleme (Load-balancing): Yük dengeleme sayesinde her bir servis kullanıcı isteklerine göre hizmet sağlamakta ve hedef servisin en uygun örneği belirlenebilmektedir. Homojen kullanıcı istekleri sıralı bir dizi mikroservisten geçerek bu istekler bir zincir halinde sunulmaktadır [33]. Ayrıca zincirlerin kullanıcı talepleri, değişen işlem süreleriyle birlikte farklı servis kalite (Quality of Service-QoS) gereksinimlerini de beraberinde getirmektedir. Çalışma [33]'de mikroservisler için mevcut yük dengeleme çözümlerinin heterojenliği veya zincirler arası rekabeti sağlayamadığı ve kullanıcı isteklerini dengelemek için zincir odaklı yük dengelemeye ihtiyaç olduğu belirtilmektedir. Ancak yazarlar, mesaj kuyruğuna sahip HTTP gibi mevcut iletişim desenlerinin mikroservis örnekleri arasında ara bağlantı yönetimini zorlaştırdığını ve zincir odaklı yük dengeleme uygulamak için ekstra zorluklar getirdiğini vurgulamaktadır.

Verileri çoğaltma (Replicating data): Mikroservisler birbirinden bağımsız olarak geliştirilebilse de bazı servislerin birbirlerine bağlı olması gerekmektedir. Bir servisteki verilerin çoğaltılması, veriler statik ise eşzamansız olarak gerçekleştirilebilmekte veya servis güncellenmemiş veriler ile de çalışabilmektedir. Ancak çoğaltma süresinin kısa olması sürecin karmaşıklığını artırmaktadır [44].

Uzak çağrılar (Remote calls): Mikroservis tabanlı sistemler her ne kadar birbirlerinden bağımsız tasarlansa da, büyük ölçekli sistemlerde karmaşık bir iletişim altyapısı bulunabilmektedir. Bu yüzden senkron yapılan uzak çağrılar, sistemin oldukça yavaşlamasına sebep olmaktadır. Çağrılan servisler, başka bir servisin verilerine veya işlevselliğine bağlıysa, basamaklı sistem arızalarına bile neden olabilmektedir [44]. Bununla birlikte çalışma [49]'da yazarlar, mikroservisler için yapılan uzak çağrıların maliyetli olduğunu belirtmektedir.

Tablolar arasındaki ilişki: Monolitik bir mimariden mikroservis tabanlı mimariye geçiş sırasında en temel zorluklardan birisi iletişim altyapısının değiştirilmesidir. Çünkü monolitik mimariler tek kod tabanlı ve tek bir ilişkisel bir veritabanı üzerinde çalıştıkları için tablolar arasındaki ilişkiler uygun şekilde tasarlanabilmekte ve ardından nesne modelleriyle eşleştirilebilektedir [50]. Tek parçalı kod tabanı mikroservislere ayırmak ve ilgili servisleri uygun veritabanlarına entegre etmek zorlu bir süreçtir. Burada tablo ile ilgili bir mantığı benimsemesi gereken her bileşen ayrı bir mikroservis haline gelmeyebilmektedir.

2.7. Mikroservislerin İletişimi ve Dağıtımı ile İlgili Araştırma Konuları ve Tespit Edilen Anahtar Bulgular

Bu tez çalışması kapsamında yapılan sistematik literatür araştırması için belirlenen araştırma sorularının amaçlarından biri, iletişim ve mikroserlerin dağıtımı alanındaki açık sorunları çözmeye yardımcı olabilecek araştırma yönergelerini belirlemektir. Mikroservis mimarilerinin kullanımı ile ortaya çıkan açık sorunlara yönelik araştırma yönergeleri aşağıda kısaca özetlenmektedir:

Karmaşıklık Kontrolü: Geliştiriciler tarafından yüzlerce mikroservisi yönetmek, kritik bir zorluk olduğu için mikroservisler için karmaşıklığı yönetebilen yararlı araçlara ihtiyaç vardır.

İzleme: Mikroservis mimarilerinde izleme mekanizmaları oldukça önem arz etmekte ve ihtiyaç haline gelmektedir. Bu izleme mekanizmaları sayesinde bir serviste oluşan bir hatadan diğer servisler etkilenmeden kısa sürede problem çözümüne ulaşılabilecektir.

Servis esnekliği: Mikroservis tabanlı bir sistemden beklenen en önemli özelliklerden birisi servislerin hataya karşı dayanıklı olması ve sürekli kullanılabilirlik için bir depolama sistemine sahip olmasıdır. Bu özellikler de servis esnekliğini beraberinde getirmektedir. Servis esnekliği sayesinde bir mikroservis, hatalı durumundan önceki kayıtlı içeriğine geri dönebilmekte ve böylece servisin başarıyla yeniden başlatılması mümkün olmaktadır.

Verimli günlük kaydı: Uygulama içerisinde meydana gelebilecek problemleri anlık olarak çözmek için etkili bir günlük kayıt tutma stratejisi gerekmektedir.

Performans iyileştirme: Mikroservislerin veri iletişimi için kullanılan iletişim modelinin performansı, irdelenmesi gereken kritik faktörlerden biridir.

Servis bağımlılıklarını azaltma: Mikroservis mimarilerinden yararlanmak için servislerin birbirlerine bağımlılığını minimize edilmesi gerekmektedir.

Hata toleransı: Mikroservisler bağımsız olarak dağıtıldığından servis örnekleri sıklıklı hataya uğrayabilmektedir. Mikroservisler arasındaki etkileşim sayısı yüksekse, bu etkileşimin, hata olması durumunda sistem üzerinde ciddi bir etkisi olabilmektedir. Bunu önlemek için, bir sistemde hata olması durumunda sistemin belirli bir memnuniyet seviyesinde çalışacak şekilde hata toleranslı olarak tasarlanması gerekmektedir.

Güvenliği sağlama: Mikroservis tabanlı sistemler genellikle bulut ortamında işlendiği için bulut bilişimden kaynaklı güvenlik tehditlerine karşı dayanıklılık sağlayan güçlü güvenlik mekanizmalarına ihtiyaç duyulmaktadır.

Otomatik dağıtım sistemleri: Servis sayısının artması, çok sayıda servisin sınırlı kapasiteye sahip kaynaklara verimli bir şekilde dağıtılmasını zorlaştırmaktadır. Mevcut dağıtım yaklaşımları kullanıcı tarafından manuel konfigürasyon dosyalarını oluşturmayı gerektirmekte ve minimum maliyeti gözetmemektedir. Bu nedenle, mikroservislerin verimli dağıtımını gerçekleştirmek için otomatik dağıtım mekanizmalarına ihtiyaç duyulmaktadır.

Araştırma yönergelerinin yanısıra mikroservisler için araştırılan iletişim ve dağıtım desenleri, bu desenlerde ortaya çıkan problemlerin değerlendirilmesi sonucunda çeşitli anahtar bulgular elde edilmiş ve bu bulgular aşağıda kısaca özetlenmektedir:

- Dağıtım ve iletişim yaklaşımları hala gelişme aşamasındadır.
- Dağıtım yaklaşımlarından en fazla kullanılan yaklaşım konteyner teknolojisidir.

- En yaygın kullanılan iletişim yaklaşımları senkron ve yayınla/abone ol (publish/subscribe) iletişim türleridir.
- İletişim ve dağıtımı desteklemek için konteyner başına servis örneği yaklaşımı sıklıkla tercih edilmektedir.
- Son yıllarda oldukça popüler olan sunucusuz dağıtım yöntemi, geliştirme ekibi tarafından dağıtım altyapısının yapılandırılmasına izin vermemesi ve zaman aşımı sınırından daha uzun sürebilen video yükleme işlemi gibi uzun süreli hizmetler için uygun olmaması gibi bazı zorluklara sahiptir.
- Hibrit iletişim metotları hem senkron hem de asenkron iletişim yaklaşımını içermektedir.
- Dağıtım için minimum maliyeti bulmak en önemli metriktir, ardından hata toleransı ve yük dengeleme gelmektedir.
- Birkaç açık araştırma zorluğu hala mevcuttur; otomatik dağıtım ve hata toleransı, önemli zorluklar olarak kabul edilmektedir.

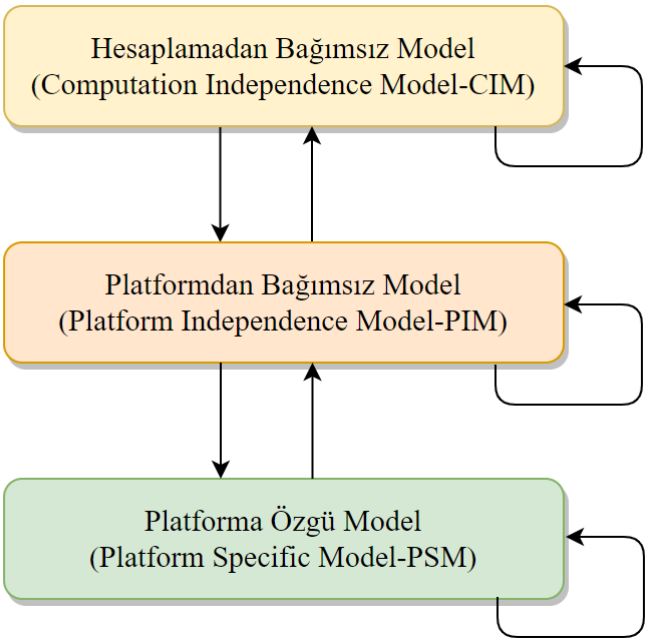
2.8. Model Güdümlü Geliştirme

Model Güdümlü Geliştirme (Model-Driven Development-MDD), yazılım geliştirmeyi daha yüksek bir soyutlama düzeyine taşıyarak yazılımın kodlama sürecine geçmeden önce model tasarımı ile gerçekleştirim zorluğunun üstesinden gelmeyi amaçlayan bir yaklaşımdır [51]. Bu yaklaşım temelde yazılımı özelleştiren modelleri ve kaynak kod elde etmek için bu modellerin dönüşümlerini kullanmaktadır. Modeller kaynak kod yazılmadan veya üretilmeden önce oluşturulmaktadır. MDD, kod tabanını görselleştirmek için modelleri kullanarak ve soyutlamayı yükseltmek amaçlı problem alanını kullanarak yazılım geliştirmeyi hızlandırmayı ve daha az maliyetli hale getirmeyi desteklemektedir. Aynı zamanda uygulama teknolojisini yazılımın iş mantığından ayırmaktadır [52]. Yazılımın işlenmesi ve mantığının uygulama teknolojisinden ayrılması, geliştiricilerin implementasyon ayrıntılarından çok problemi çözmeye odaklanmasını sağlamaktadır [52].

MDD'nin arkasındaki temel fikir bir sistemi öncelikle modeller üzerinden tasarladıktan sonra gerçek bir yapıya dönüştürmektir. Sistemler, çeşitli soyutlama düzeylerinde ve birden çok perspektifte modellenebilmektedir. Oluşturulan modeller, yazılım geliştirmenin temel eserleridir [53]. Bu modeller geliştirildikten sonra üreticiler kullanılarak veya çalışma zamanında modeller çalıştırılarak hizmet veren sistemlere dönüştürülmektedir [54].

Object Management Group (OMG) tarafından ortaya çıkarılan Model Güdümlü Mimari (Model-Driven Architecture-MDA) MDD'nin en bilinen örneğidir [51, 55]. OMG [55], MDD'nin uygulanması için standartlar (UML, MOF, XMI, CWM) geliştiren endüstri odaklı bir kuruldur. MDA'da kodu görselleştirmek için UML (Unified Modelling Language) diyagramları kullanılmaktadır [56].

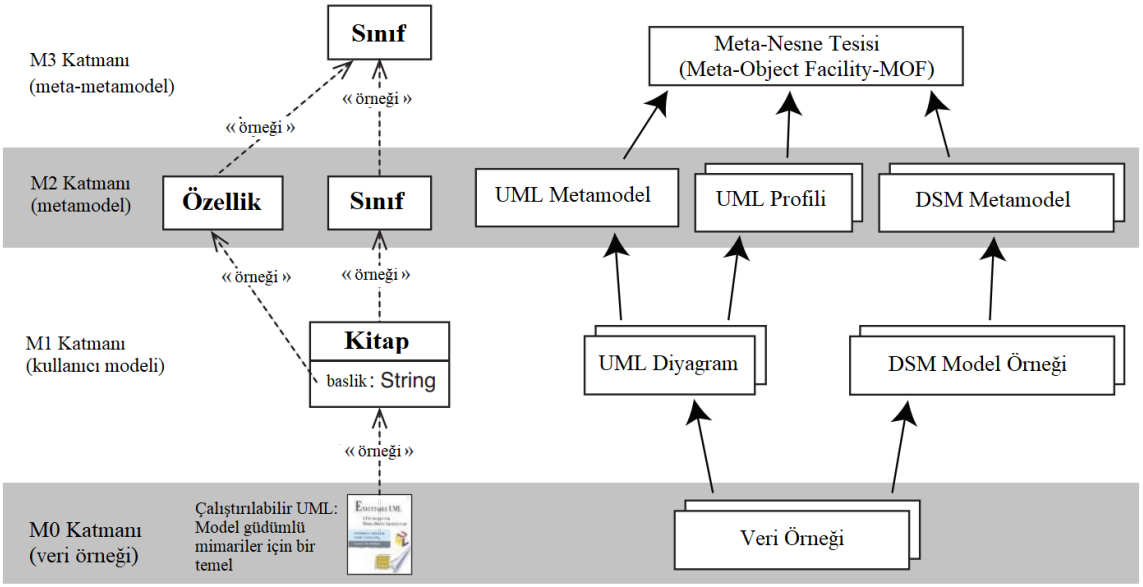
MDA (1) taşınabilirlik, (2) birlikte çalışabilirlik ve (3) yeniden kullanılabilirlik olmak üzere üç temel hedefe odaklanmaktadır ve bu hedeflere ulaşmak için anahtar soyutlama terimi "endişelerin mimari ayrımı (architectural separation of concerns)" dır [1]. MDA, Hesaplamadan Bağımsız (Computation Independent), Platformdan Bağımsız (Platform Independent) ve Platforma Özgü (Platform Specific) olmak üzere bakış açıları (viewpoints) adı verilen üç ana mimari soyutlama katmanını tanımlamaktadır [54]. Hesaplamadan Bağımsız Model (Computation Independent Model-CIM) sistem alanındaki uygulayıcıların aşina olduğu terminolojiyi kullanarak bir sistemin çalışacağı ortamı ve sistem gereksinimlerini açıklamaktadır. Platformdan Bağımsız Model (Platform Independent Model-PIM), sistem farklı platformlarda kullanıldığında değişmesi muhtemel olmayan özelliklerle ilgilenmektedir [51]. Platforma özgü uygulama ayrıntılarını belirtmeden bir sistemin yapısını ve işlevlerini resmi olarak açıklamaktadır [1]. MDA'nın en düşük seviyesinde yer alan Platforma Özgü Model (Platform Specific Model-PSM) verilen bir platform üzerinde sistemin uygulanması için gerekli detayları içermektedir. MDA'da platform denildiğinde bir sistemin üzerinde çalışabileceği Sun'ın Java EE veya Microsoft'un .NET platformları gibi uyumlu alt sistemler ve teknolojiler kümesi kastedilmektedir. Şekil 2.3.' de MDA mimari katmanları ve model dönüşümleri gösterilmektedir.



Şekil 2.3. MDA mimari katmanı ve model dönüşümleri [1]

Şekil 2.3.' da görüldüğü gibi model eşlemeleri ve model dönüşümleri MDA'nın anahtar görünümüdür [1]. Çizilen her bir ok işareti bir modelden diğer modele dönüşümü temsil etmektedir. Model dönüşümleri bir çok amaçla ve birçok seviyede gerçekleşebilmektedir. Dönüşümler genellikle çift yönlü olup, modeller arasında yukarı ve aşağı eşleşmeler yapılabilmektedir. CIM-CIM veya PIM-PIM gibi aynı tür modeller üzerindeki eşlemeler, bir analiz aşamasından bir tasarım aşamasına [57] geçerken meydana gelen dönüşüm gibi model iyileştirmelerini temsil etmektedir.

MDA Kılavuzu [58] çok çeşitli dönüştürme türlerini, tekniklerini ve modellerini tartışmaktadır. Genel olarak MDD'de olduğu gibi, model dönüşümleri kavramı MDA felsefesinin merkezinde yer almaktadır. Şekil 2.4.'de dört adet MDA modelleme ve metamodelleme katmanı görülmektedir. Bu katman aşağıdaki gibi tanımlanmaktadır [1].



Şekil 2.4. MDA modelleme ve metamodelleme katmanları [1]

- M3: M2 seviyesinin modeli olan meta-metamodel katmanı, bir metamodel içerisinde görünen Sınıf gibi kavramları açıklamaktadır. Örneğin UML metamodelinin metamodeli MOF [59] M3 katmanını tanımlamaktadır. Eclipse modelleme ortamının metamodeli bu çalışmada kullanılan Ecore [60] metamodelidir. ISIS Generic Modeling Environment (GME) aracının metamodeli metaGME [61]'dir.

- M2: M1 seviyesinin modeli olan metamodel katmanı; bir modelleme dilini oluşturan kavramları açıklamaktadır. Bir UML metamodeli, Executable (Çalıştırılabilir) UML profili ve belirli bir şirket veya endüstri segmenti için oluşturulmuş ve özelleştirilmiş alana özgü bir metamodel bu katmanda oluşturulabilmektedir.

- M1: M0 seviyesinin modeli olan kullanıcı modeli veya model örneği katmanı; sınıf diyagramları, durum şemaları ve diğer bu tür yapılar M1 katmanı öğeleridir.

- M0: Veri örneği katmanı; nesneler, kayıtlar, veri ve ilgili eserler bu seviyede mevcuttur.

2.9. Eclipse Modelleme Aracı (Eclipse Modeling Tool)

2.9.1. Veri Modeli

Domain model olarak da adlandırılan veri modeli, üzerinde çalışmak istenilen veriyi temsil etmektedir. Örneğin çevrimiçi bir uçuş rezervasyon uygulaması geliştirildiğinde Müşteri, Uçuş, Rezervasyon gibi nesneler, EMF aracı ile UML diyagram şeklinde oluşturularak bir domain model üretilebilmektedir. Bu araç sayesinde uygulama mantığından ve kullanıcı arayüzünden bağımsız bir şekilde bir uygulamanın veri modeli tasarlanabilmektedir.

EMF, nesneler oluşturmak için görsel bir arayüz oluşturarak modelin net bir şekilde görülmesini sağlamaktadır. Aynı zamanda bu model üzerinden kod üreteci ayarlanarak istenilen zamanda modelden Java kodu üretilebilmektedir.

2.9.2. Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF), veri modelini tasarlamak ve modele bağlı olarak kod ve diğer çıktıları üretmek için kullanılan bir Eclipse plugin kümesidir. EMF, meta-model ile gerçek model arasında bir ayrıma sahiptir. Meta-model modelin yapısını tanımlamaktadır. Bir model bu meta modelin somut bir örneğidir. EMF geliştiricinin XMI, Java açıklamaları, UML veya bir XML şeması gibi farklı anlamlarda metamodeller oluşturmasına izin vermektedir. Aynı zamanda varsayılan uygulama, XML Metadata Interchange adlı bir veri biçimini kullanarak model verisinin sürekliliğini sağlamaktadır.

2.9.3. Bir EMF Modelden Veri Üretme

EMF modellerde saklanan bilgiler, türetilmiş bir çıktıyı üretmek için kullanılmaktadır. Tipik bir kullanım senaryosunda geliştirici uygulamasının domain modelini tanımlamak ve bu modelden karşılık gelen Java implementasyon sınıflarını üretmek için EMF'yi kullanmaktadır. EMF çerçevesi, oluşturulan kod üzerinde geliştiriciye güvenli bir şekilde modeli genişletip

değiştirme hakkı vermektedir. EMF model aynı zamanda HTML sayfaları, ya da uygulamanın çalışma zamanında yorumlanacağı farklı çıktılar üretmek için kullanılabilmektedir.

2.9.4. Meta modeller - Ecore ve Genmodel

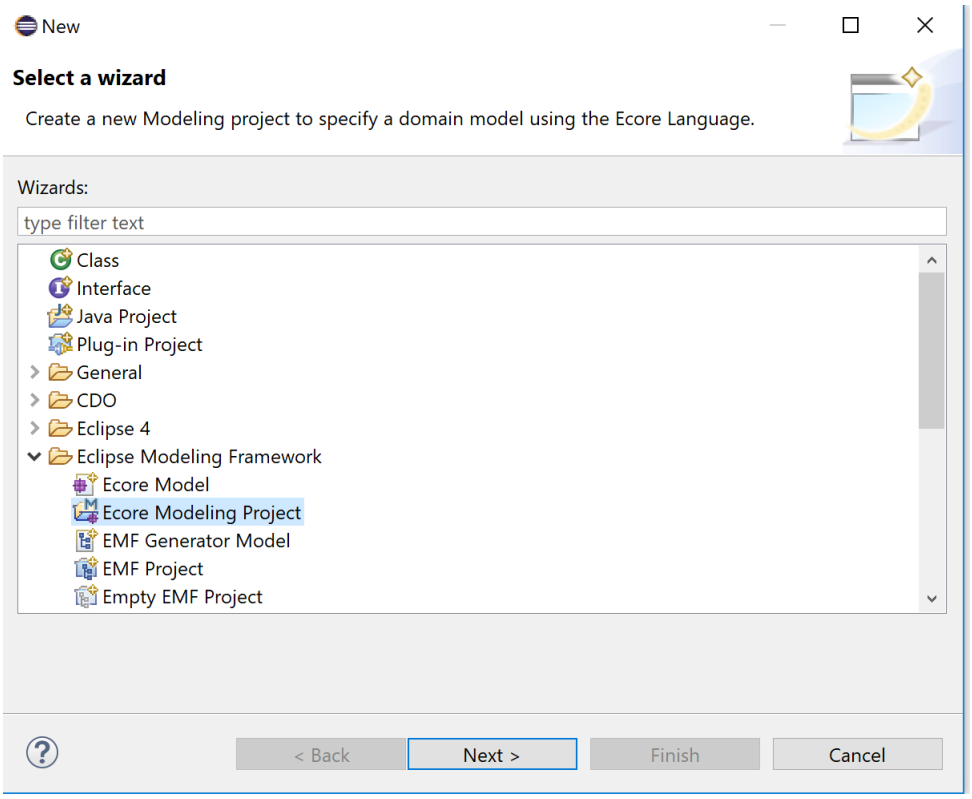
EMF metamodel, ecore ve genmodel tanımlama dosyaları olmak üzere iki parçadan oluşmaktadır. Ecore dosyası tanımlanan sınıflar hakkında bilgiler içermektedir. genmodel dosyası ise yol ve dosya bilgileri gibi kod üretimi için ek bilgileri içermektedir. genmodel aynı zamanda kodun doğru bir şekilde üretilmesini sağlayan kontrol parametrelerini içermektedir.

2.9.5. Ecore Açıklama Dosyası

Bir ecore dosyası aşağıdaki bileşenleri içermektedir:

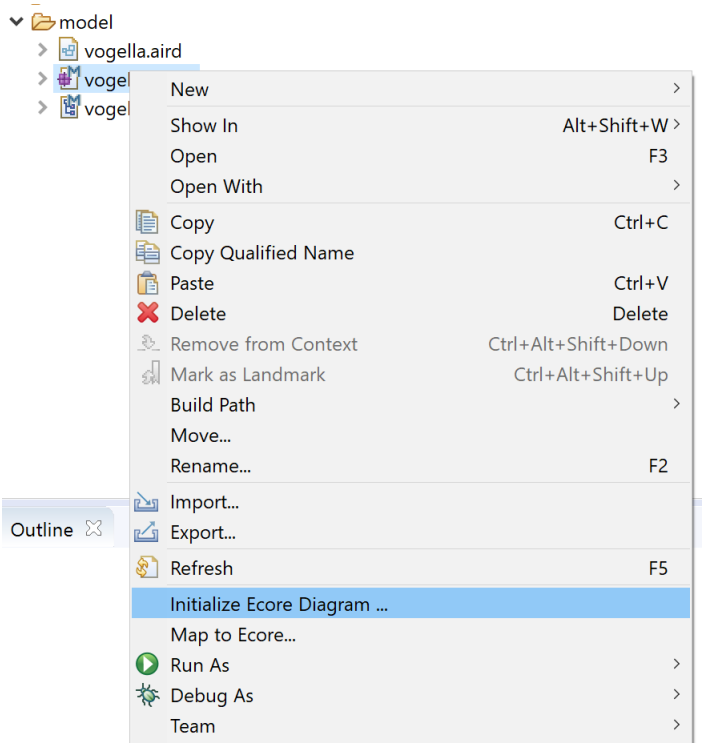
- EClass: Sıfır ya da daha fazla özellik ve metot içeren bir sınıfı ifade etmektedir.
- EAttribute: Bir adı ve tipi olan özelliği ifade etmektedir.
- EReference: Sınıflar arasındaki ilişkiyi kurmak için kullanılmaktadır. 1'e 1, 1'e çok, çoka 1 gibi temsil ettiği birçok bağlantı türü bulunmaktadır.
- EDataType: int, float ya da java.util.Data gibi javada kullanılan değişken türlerini içermektedir. EMF modellemede özelliğin türünü ifade etmektedir.

Ecore modeli tüm modeli temsil eden bir kök nesnesini göstermektedir. Bu model bir paket dosyasını içermekte, paket dosyası içerisinde de sınıflar bulunmaktadır. Sınıflar içerisinde ise tanımlanan özellikler ve metotlar, sınıfların çocuğu olarak temsil edilmektedir.



Şekil 2.5. Eclipse modelleme projesi oluşturma

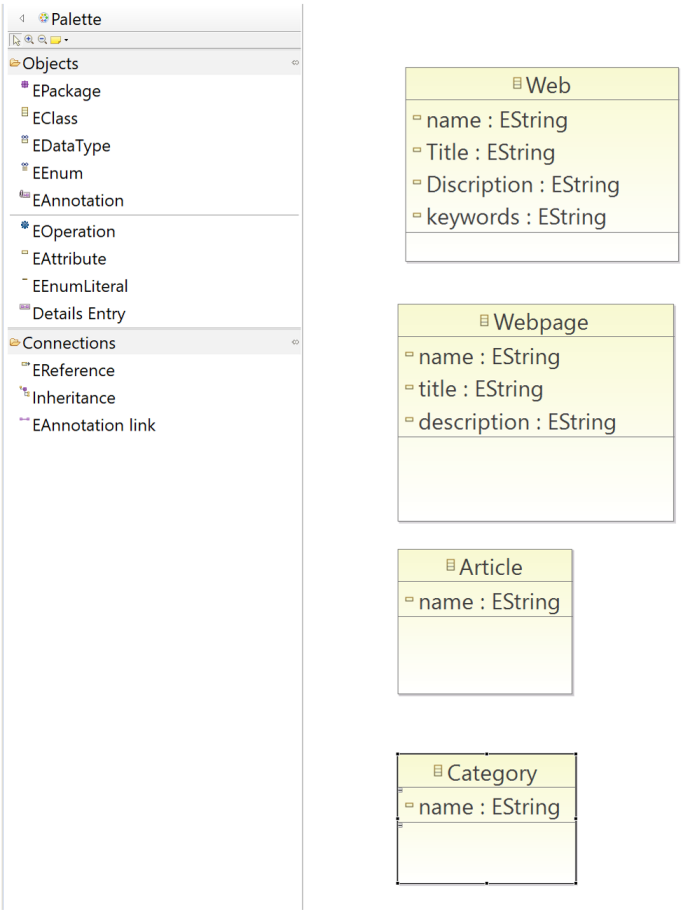
Eclipse'de Şekil 2.5.'deki gibi Eclipse Modeling Project oluşturulduktan sonra .ecore dosyasına sağ tıklanarak Initialize Ecore Diagram seçeneği seçildiğinde mevcut ecore modelinin grafiksel gösterimi Şekil 2.6.'daki gibi oluşturulabilmektedir.



Şekil 2.6. Ecore diyagramı başlatma

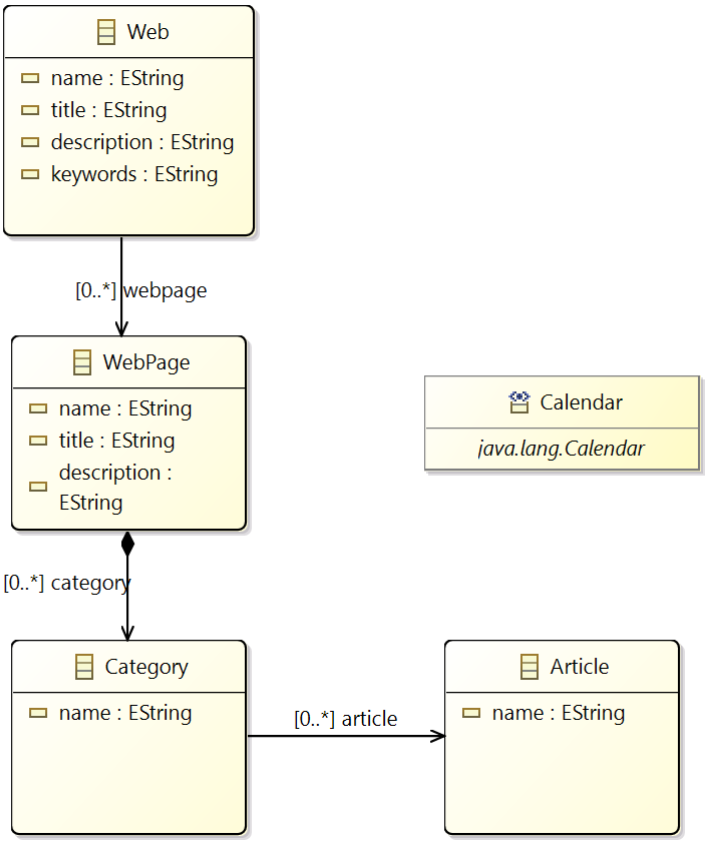
2.9.6. EMF Model Oluşturma ve Modelden Kod Üretimi

Eclipse Modeling Project üzerinden .ecore uzantılı bir domain modele isimlendirme yapılmaktadır. Şekil 2.7.'deki gibi modele webpage adı verildikten sonra ekrana gelen görsel editörde Class, Relation, Attribute gibi birçok bileşen yer almaktadır.



Şekil 2.7. EMF görsel editörü

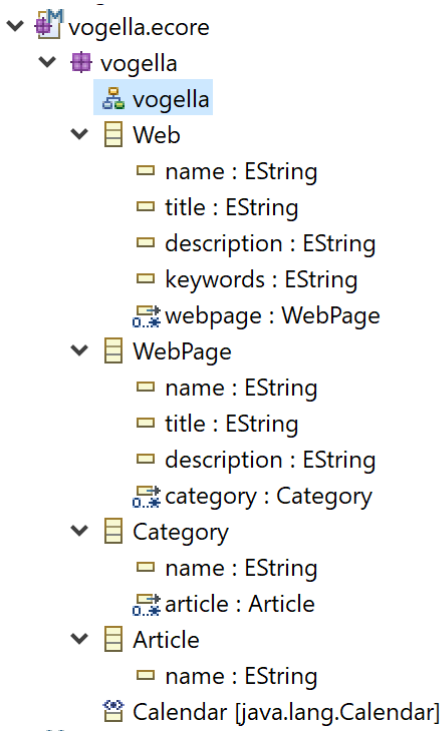
Şekil 2.8.'de görüldüğü gibi EClass, EType, EAttribute ve EReference kullanılarak bir web arayüzünün küçük bir parçasını ifade eden bir ecore model oluşturulabilmektedir. Bu bileşenlerin her birinin özelliği Properties kısmından görülebilmektedir.



Şekil 2.8. Ecore diyagram bileşenleri oluşturma

2.9.7. Ecore Diyagramı Görüntüleme

Diyagram oluşturulup kapatıldıktan sonra sol pencereden webpage.ecore dosyası açılmakta ve Şekil 2.9.'da görüldüğü gibi diyagramda oluşturulan tüm bileşenler anne-çocuk ilişkisi şeklinde görülmektedir. Sınıfların altında metotlar ve özellikler yer almaktadır.

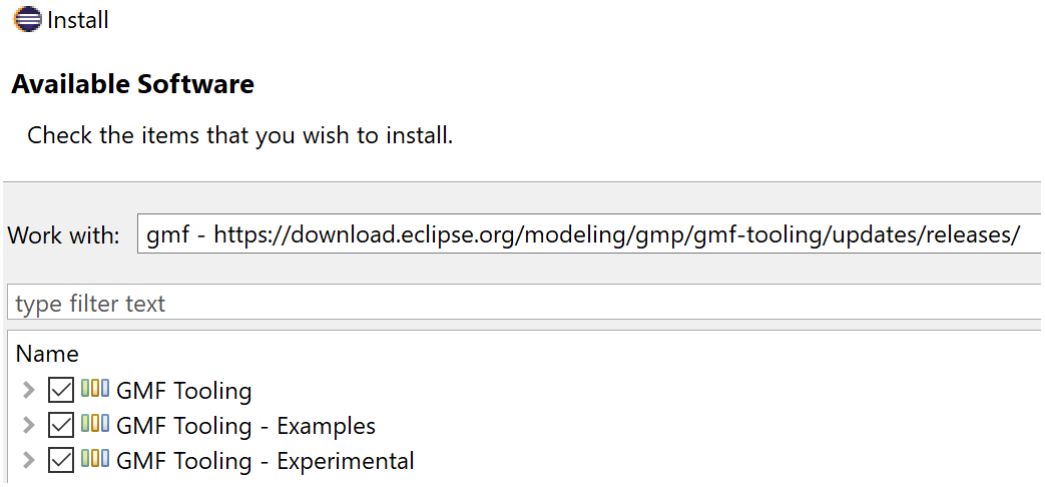


Şekil 2.9. Ecore diyagram bileşenleri ve özellikleri

2.10. Grafiksel Modelleme Çerçevesi (GMF)

Grafiksel modelleme çerçevesi (GMF) Eclipse Modeling Framework (EMF) ve Graphical Editing Framework (GEF) arasında üretici bir köprü sağlamayı amaçlayan bir Eclipse modelleme projesidir. Eclipse IDE 2018 sürümü ile ecore modeli oluşturmak kontrol paneli (dashboard) aracılığı ile domail model, domain gen model, graphical def model, tooling def model, mapping model ve diagram editor gen model kolaylıkla türetilmektedir.

Bu bölümde mindmap uygulaması üzerinden version 2.0.1'de GMF tarafından sağlanan fonksiyonellikler açıklanmaktadır. GMF aracının Eclipse projesinde kullanılması için öncelikle Help → Install New Software sekmesinden Şekil 2.10.'da görüldüğü gibi https://download.eclipse.org/modeling/gmp/gmf-tooling/updates/releases/ URL'i girilerek GMF aracı için gerekli yazılımlar indirilmektedir.

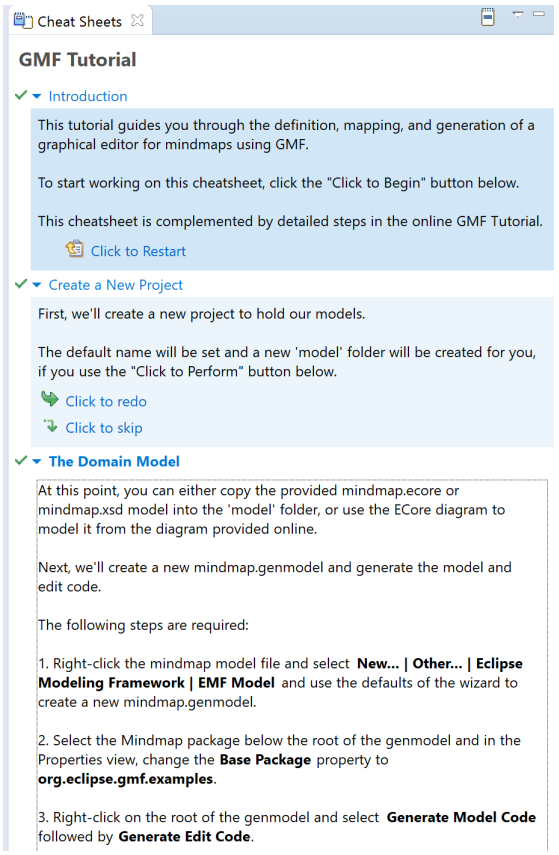


Şekil 2.10. GMF aracı için gerekli yazılımların indirilmesi

2.10.1. Örnek GMF Uygulaması (Mindmap Editörü)

GMF aracı için gerekli yazılımlar kurulduktan sonra File → New → Other → Graphical Modeling Framework → Graphical Editor Project seçilip boş bir GMF projesi oluşturulabilmektedir. Daha sonra otomatik olarak klasörün içerisinde bulunan model dosyasına .ecore uzantılı GMF dokümantasyonunda yer alan örnek mindmap dosyası [62] referansı üzerinden indirilip eklenmektedir. Bu dosyada Topic, Map, Resource, Relationship gibi bir çok sınıf oluşturularak sınıflara ait gerekli olan özellikler tanımlanmaktadır.

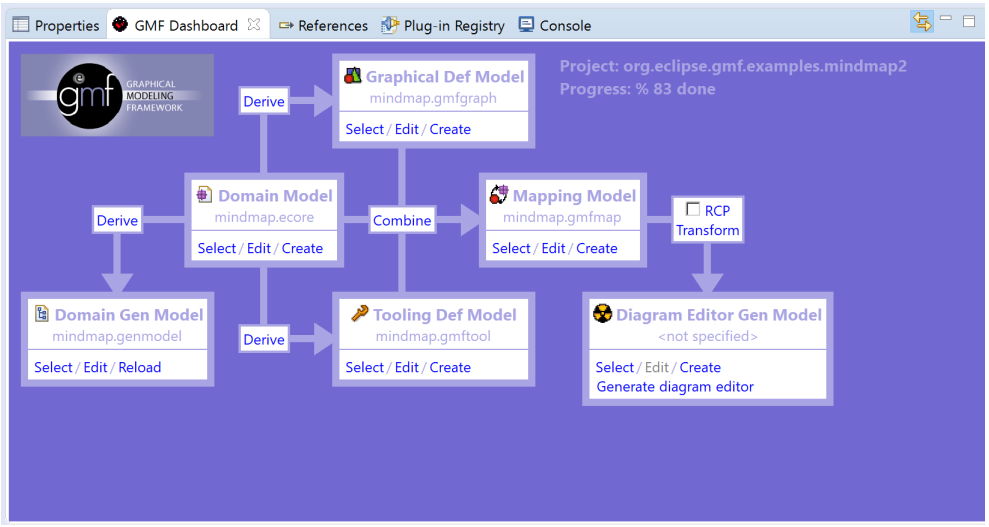
Proje oluşturulduktan sonra adım adım yapılması gerekenler hakkında bilgi veren Şekil 2.11.'deki gibi bir bilgilendirme alanı Eclipse ekranının sağ köşesinde yer almaktadır.



Şekil 2.11. GMF kullanımı için Eclipse'in sunduğu kopya kağıdı

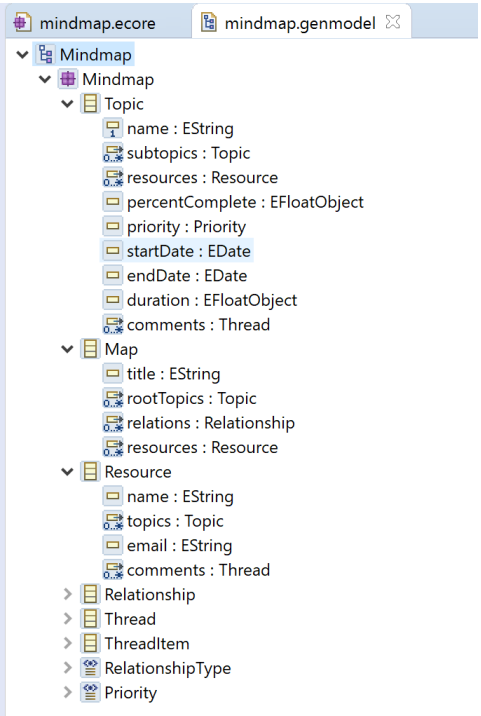
2.11. Domain Gen Model Oluşturma

Mindmap.ecore dosyası oluşturulduktan sonra bu dosya kullanılarak Domain gen model oluşturma işlemine geçilmektedir. mindmap model dosyasına sağ tıklanıp New–> Other–> .. Eclipse Modeling Framework –> EMF Model seçilip mindmap.genmodel oluşturmak için gerekli varsayılanlar kullanılmaktadır. Ayrıca tüm bu işlemleri yapmak yerine GMF Dashboard üzerindeki Domain Model'den Domain Gen Model'e uzanan "derive" linkine tıklanarak da .genmodel uzantılı dosya oluşturulabilmektedir. GMF Kontrol Paneli'nin ekran görüntüsü Şekil 2.12.'de verilmektedir.



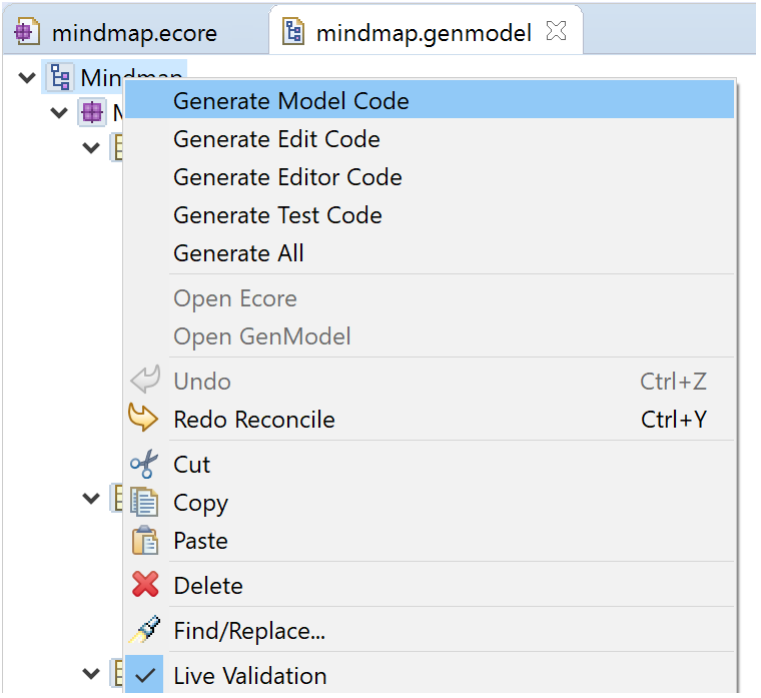
Şekil 2.12. GMF Kontrol Paneli (GMF Dashboard)

Oluşturulan mindmap.genmodel dosyasının ekran görüntüsü Şekil 2.13.'teki gibidir. mindmap.genmodel dosyası oluşturulduktan sonra genmodel kökünün altındaki Mindmap paketi seçilip Özellikler sekmesi görüntülenerek Base Package kısmına proje dosyasının adı (bu örnek için org.eclipse.gmf.examples) yazılmaktadır.



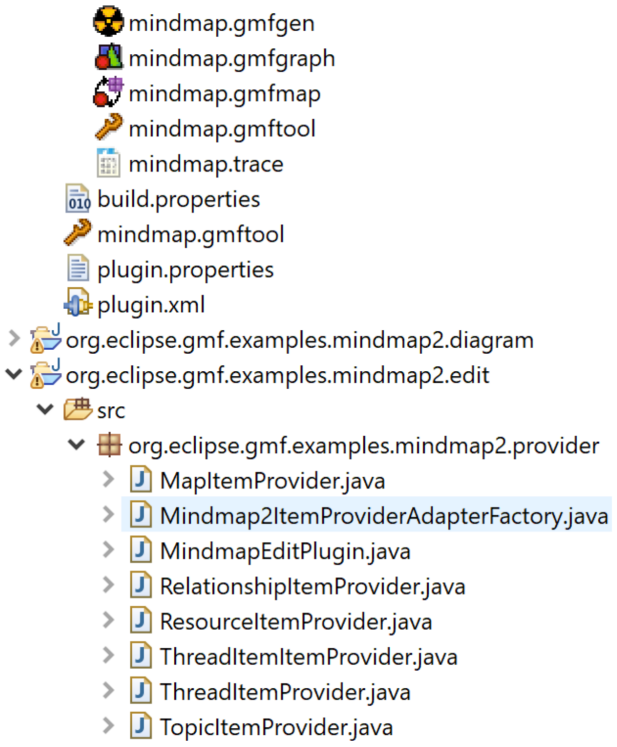
Şekil 2.13. mindmap.genmodel dosyası

Bu adımdan sonra genmodel köküne sağ tıklanarak Şekil 2.14.'daki gibi "Generate Model Code" ve ardından "Generate Edit Code" seçilmektedir.



Şekil 2.14. Gen modelden kod üretme

Generate Model Code seçeneği seçildiği zaman ana klasör içerisinde build.properties, plugin.properties ve plugin.xml dosyaları oluşmaktadır. Generate Edit Code seçeneğinde ise Şekil 2.15.'da görüldüğü gibi "klasör ismi".edit şeklinde bir dosya oluşarak src dosyası içerisinde mindmap.ecore dosyasında yer alan tüm sınıfların sağlayıcı dosyası (provider) oluşturulmaktadır.

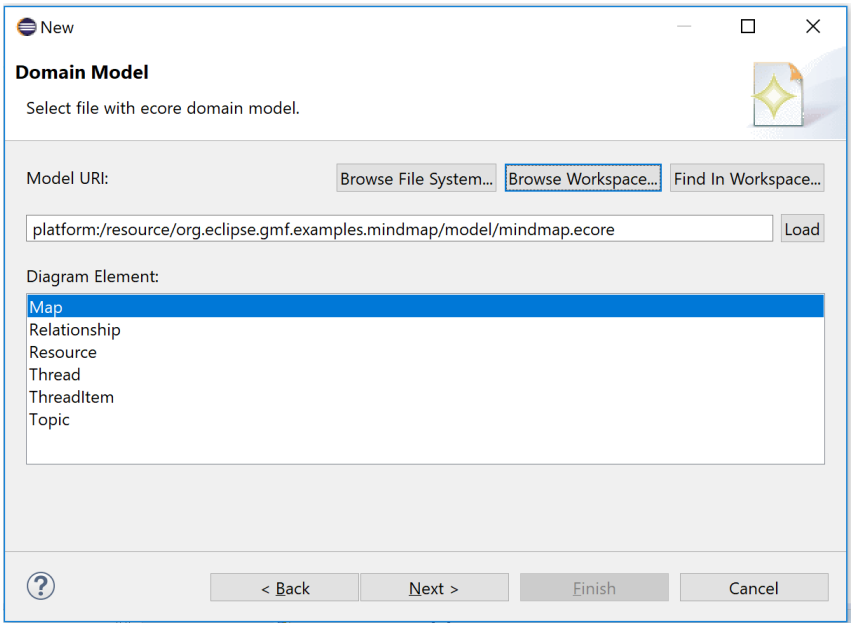


Şekil 2.15. Generate Model Code ve Edit Code seçenekleri ile oluşan dosyalar

Bu aşamadan sonra mindmap uygulaması için grafiksel (graphical) ve haritalama (mapping) tanımlamaları oluşturulmaya başlanmaktadır.

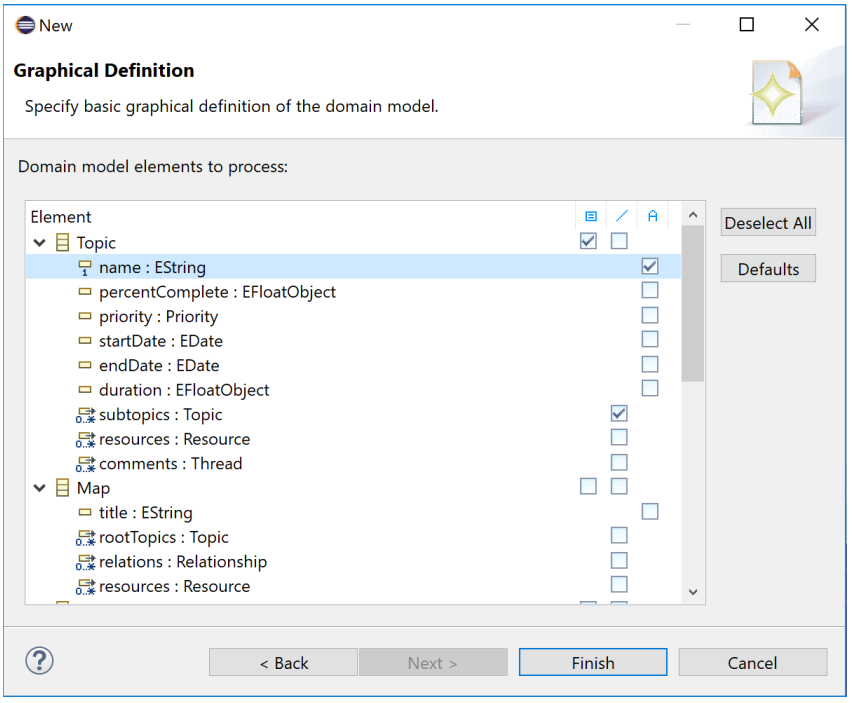
2.11.1. Grafiksel Tanımlama (Graphical Definition)

Oluşturulan diyagram üzerinde gösterilen şekiller, düğümler ve bağlantıları tanımlamak için grafiksel bir tanım modeli kullanılmaktadır. Grafiksel bir tanımlama oluşturmak için kopya kağıdında belirtilen ve yapılması gereken ilk işlem mindmap.gmfgraph model oluşturmak için klasör ismi/model'ni parent (ana) dosya olarak seçmek gerekmektedir. Daha sonra Domain model ve Graphical Def Model arasında kalan "derive" linki tıklanarak .gmfgraph uzantılı dosya oluşturulmaktadır. Next seçeneğinden sonra Şekil 2.16.'da görüldüğü gibi Diyagram elementi olarak kök sınıf olan Map seçilmektedir.



Şekil 2.16. Grafiksel model tanımlama

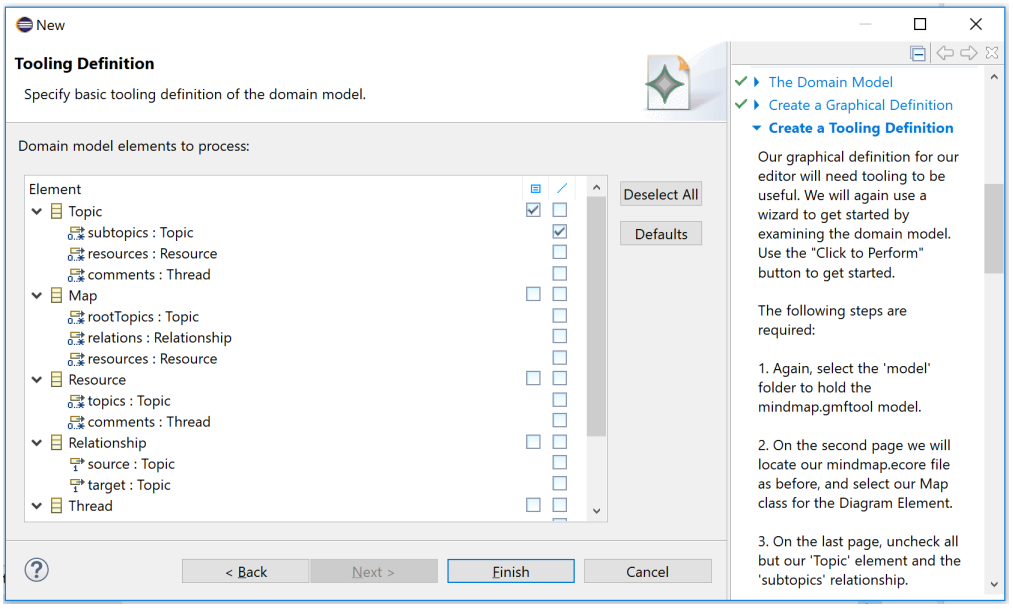
Son sayfada domain modelde bulunan özellikler yer almakta ve bunları seçme şansı verilmektedir. Mindmap örneğinde Şekil 2.17.'deki gibi "Topic" elementi bir düğüm olarak,"name" bir özellik olarak "subtopic" ise bir ilişki olarak seçilmektedir.



Şekil 2.17. Grafiksel model tanımlama için kullanılan element ve özellikler

2.11.2. Kalıp Tanımlama (Tooling Definition)

Oluşturulan grafiksel tanımlamanın kullanılabilir olması için bir kalıba ihtiyacı vardır. Grafiksel öğeler oluşturmak ve palet, araç oluşturma, eylemler gibi işlemleri belirtmek için gerekli bir modeldir. Bunun için model klasörü tıklanarak GMF Dashboard üzerinde yer alan Domain Model ve Tooling Def Model arasındaki "derive" linkine tıklanarak . gmftool uzantılı dosya oluşturulmaktadır. İkinci sayfada grafiksel tanımlamada olduğu gibi Diyagram Element için Map sınıfı seçilmektedir. Son sayfada ise Şekil 2.18.'de görüldüğü gibi Topic elementi ve subtopics ilişkisi dışında diğer seçeneklerin onayı kaldırılmaktadır. Böylece araç paletinde yalnızca Topic sınıfı ve subtopics ilişkisi kullanılabilecek hale gelmektedir. Tasarımcının ihtiyacına göre özellikler, sınıflar ve ilişkiler seçilebilmektedir.



Şekil 2.18. Kalıp tanımlama için element ve ilişkilerin belirlenmesi

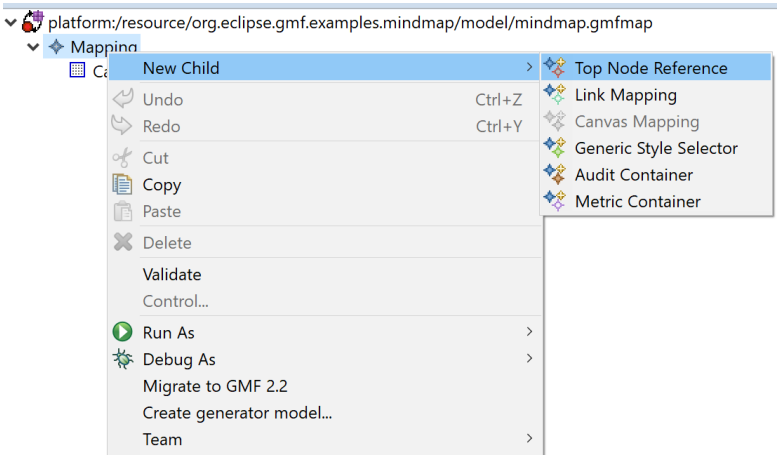
2.11.3. Haritalama Tanımlama (Mapping Definition)

Haritalama tanımlama modeli şuana kadar oluşturulan domain, grafiksel tanımlama ve kalıp tanımlama modellerinin birleştirilmesi ile oluşturulan bir modeldir. Bu modeli oluşturmak için diğer modellerde olduğu gibi projede yer alan model klasörü seçilerek GMF Dashboard

üzerinde yer alan Domain model ve Mapping model arasında yer alan "combine" linkine tıklanarak mindmap.gmfmap dosyası oluşturulmaktadır. İkinci sayfada ise mindmap.ecore modeli seçili olarak gelecektir. Next butonuna tıklandıktan sonra canvas için Map sınıfı seçilmekte ve üçüncü sayfada mindmap.tool modelinin halihazırda yüklendiği görülmektedir. Varsayılan ayarlar değiştirilmeden Next seçeneğine tıklanmaktadır. Dördüncü sayfada mindmap.gmfgraph modelinin yüklendiği görülmektedir. Son sayfada Topic ve subtopics dışında diğer tüm onaylar kaldırılarak Finish butonu ile haritalama işlemi sonlandırılmaktadır.

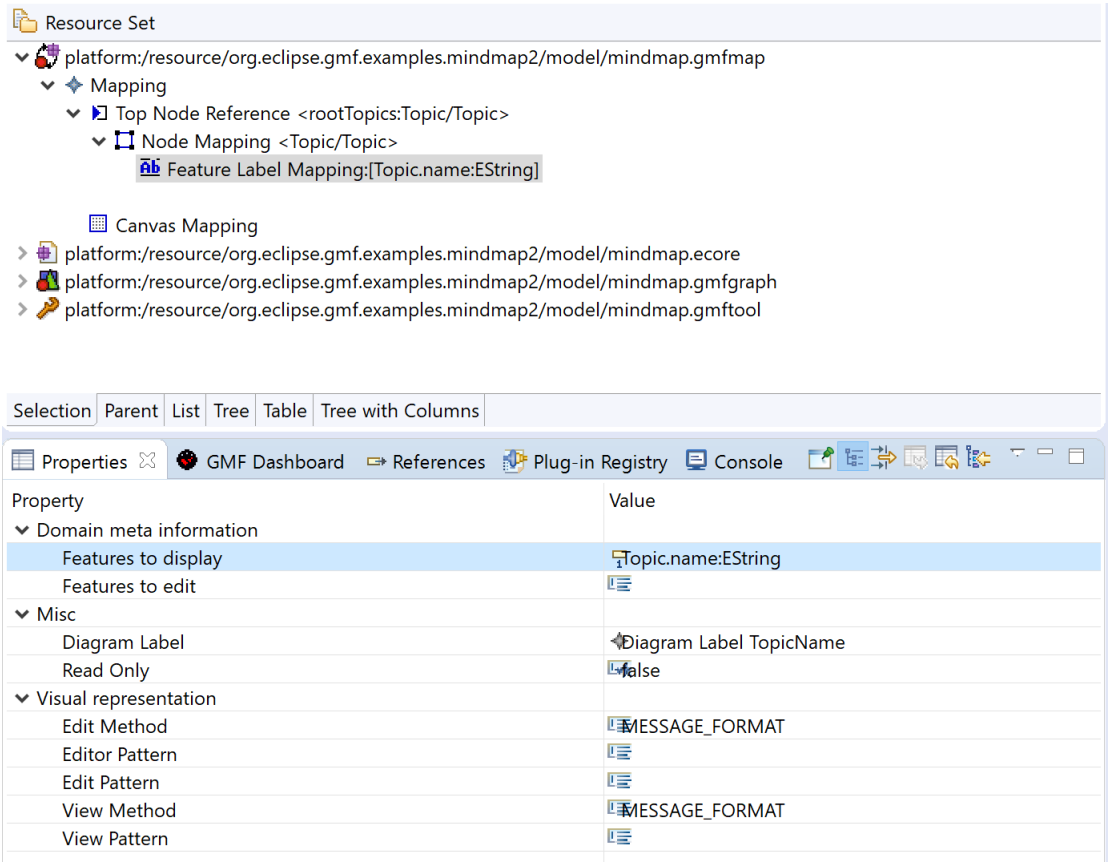
2.11.4. Etiket Haritalama (Label Mapping)

Tüm bu işlemlerden sonra manuel olarak yapılması gereken şey grafiksel tanımlamadan tasarımda kullanılacak olan elemanların özelliklerini oluşturmaktır. Bunun için mindmap.gmfmap üzerinde oluşturulan Mapping canvas dosyasına Şekil 2.19.'da görüldüğü gibi sağ tıklanarak Top Node Reference eklenmektedir.



Şekil 2.19. Etiket oluşturma

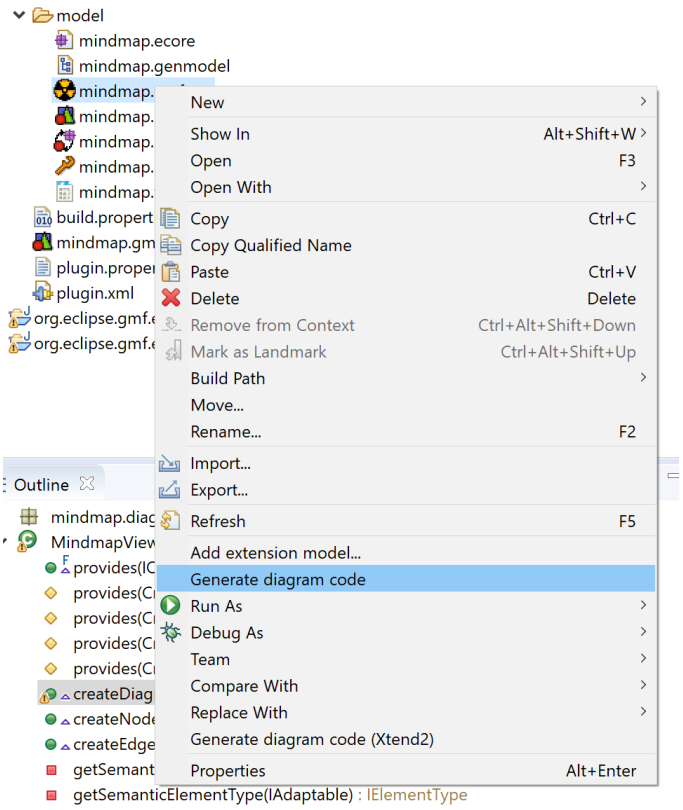
İkinci aşama olarak Top Node Reference altında oluşan Node Mapping'e sağ tıklanıp Feature Label Mapping seçilmektedir. Şekil 2.20.'de görüldüğü gibi Properties kısmından diyagram etiketi olarak TopicName seçilmektedir. Features kısmında ise "name" özelliği seçilmektedir.



Şekil 2.20. Özellik etiketi belirleme

2.11.5. Kod Üretme (Code Generation)

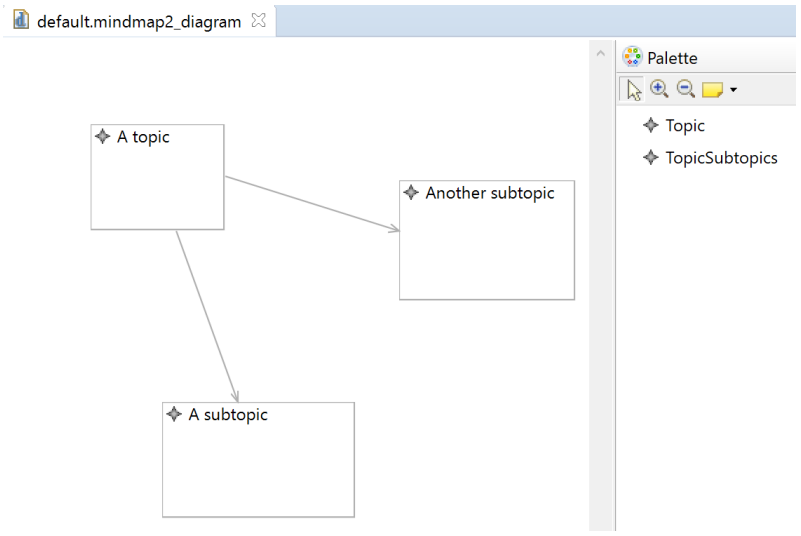
Metamodelleme için son aşama olan kod üretme modeli, haritalama modelinden dönüştürülmekte ve oluşturulan editörü temsil eden kaynak kodu oluşturmak için gerekli özellikleri içermektedir. Üretici modeli oluşturmak için Şekil 2.21.'de görüldüğü gibi mindmap.gmfgen dosyasına sağ tıklanarak Generate Diagram Code seçilmektedir.



Şekil 2.21. Diyagram kodu oluşturma

2.11.6. Diyagramın Çalıştırılması

Generate Diagram Code seçeneği ile Eclipse içerisinde .diagram uzantılı bir dosya otomatik olarak oluşmaktadır. Bu dosyaya sağ tıklanıp Run as → Eclipse Application seçilmektedir. Böylece yeni bir Eclipse programı üzerinde oluşturulan modelin testi için yeni bir proje oluşturulmakta ve New → Examples → Mindmap Diagram seçilmektedir. Önceki başlıklarda gerçekleştirilen işlemler sonrası Mindmap Diagramının Examples kısmında görülmesi gerekmektedir. Görülmediği takdirde aşamaların birinde bir yanlışlık olduğu anlaşılmaktadır. Diyagram dosyası seçildikten sonra oluşturulan test klasörü içerisinde yer alan diagram uzantılı dosyada artık Topic elementi ve subtopics ilişkisi kullanılarak metamodel oluşturma işlemi yapılabilmektedir. Şekil 2.22.'de örnek olarak bir diyagram oluşturulmuştur.



Şekil 2.22. Topic elementi ve subtopics linki ile basit bir diyagram tanımı

3. İLİŞKİLİ ÇALIŞMALAR

Mikroservislerin bulut ortamında sunuculara tahsis edilmesi işleminin verimli bir şekilde gerçekleştirilmesi, hem bulut ortamının etkin kullanımı hem de mikroservis tabanlı bir sistemin performanslı bir şekilde çalışmasını sağlamaktadır. Günümüzde mikroservislerin kaynaklara dağıtımı genellikle sistemi çok iyi bilen bir uzman tarafından salt beyin gücü ile yapılmaktadır. Bu da sistemdeki mikroservis sayısı arttıkça yüzlerce hatta binlerce mikroservisin manuel olarak en iyiye yakın bir şekilde yerleştirilmesini zorlaştırmaktadır. Otomatik yerleştirme yapan araçlar incelendiğinde ise Kubernetes, Docker Swarm, Apache Mesos ve diğer alternatif teknolojiler, bir kullanıcının oluşturduğu konfigürasyon dosyasındaki talimatlara göre dağıtım işlemini yapmaktadır. Fakat bu teknolojilerde verilen konfigürasyon dosyasına göre en iyiye yakın çözüm bulunamadığında, kullanıcının konfigürasyon dosyasını yeniden oluşturması gerekmektedir. Ayrıca servislerde oluşan yoğun trafiğin dinamik olarak kullanıcı tarafından ele alınması zorlaşmakta ve dağıtım için yapılan işlemler bir algoritmik yaklaşıma göre oldukça uzun sürebilmektedir. Bu durum da mikroservislerin en verimli şekilde kısıtlı kaynaklara yerleştirilmesini engellemektedir. Bu bölümde mikroservislerin kısıtlı kaynaklara verimli şekilde dağıtımını ele alarak yeni bir yaklaşım öneren çalışmalar, mikroservis mimarilerinin tasarım aşamasında incelenmesi için model güdümlü geliştirme yaklaşımını kullanan araç destekli çalışmalar ve dağıtım problemine algoritmik olarak yaklaşan çalışmalar alt başlıklar halinde incelenmektedir.

3.1. Model Güdümlü Mühendislik Yaklaşımı Kullanan Çalışmalar

Literatürde mikroservisler için geliştirilen araç destekli çalışmalar incelendiğinde model-güdümlü mühendislik kullanılarak mikroservislerin kullanıcı tarafından oluşturulup sistemin analiz edilebileceği az sayıda çalışma bulunduğu görülmektedir. Bunlardan Granchelli vd. [63] model-güdümlü mühendislik prensiplerine bağlı olarak MicroART isimli bir mikroservis mimarisi geri kazanma (recovery) aracı sunmaktadır. Bu araç, Fiziksel Mimari Kurtarma (Physical Architecture Recovery), Hizmet Keşfi Tanımlama (Service Discovery Identification) ve Mantıksal Mimari Kurtarma (Logical Architecture Recovery) olarak üç aşamada

sistemi kurtarmaktadır. Çalışmada mikroservis mimarisinin model tabanlı gösterimine olanak tanındığı için bu araç tarafından üretilen modellerin grafik olarak oluşturulabilmesi sağlanmaktadır. Ayrıca dokümantasyon, mimari analiz, mimari akıl yürütme veya yerleştirilen mimari ile tasarlanan mimari arasındaki doğrulama gibi bir çok amaç için kullanıldığı belirtilmektedir. Yapılan literatür araştırmalarına göre bu çalışma, mikroservis mimarilerini model tabanlı mühendislikte kullanan nadir çalışmalardan biridir fakat geliştirilen araç dağıtım teknolojisi olarak Docker kullanmış olup sadece kurtarma üzerine bir yaklaşım geliştirmektedir. Benzer şekilde model-güdümlü mühendislik kullanan bir başka çalışmada [64] mikroservis mimarilerindeki performans ve dayanıklılık mühendisliği yaklaşımlarını kıyaslamak için temel bir metamodel, bir üretim platformu ve iş yükü üretimi, sorun enjeksiyon ve izleme için destek hizmetleri içeren üretken bir platform geliştirilmektedir. Geliştirilen araçta mikroservislerin dağıtımı için Kubernetes konteyner teknolojisi kullanılmıştır ve amaç, istenilen özelliklere sahip mikroservis mimarilerinin performans ve dayanıklılık mühendisliği yaklaşımlarının ölçüm tabanlı değerlendirilmesidir. Araç üzerinde üretilen mikroservis ortamına sorunlar enjekte edilirken servislerin davranışları değerlendirilebilmekte, böylece araç anomali tespitine olanak tanımaktadır. Bu yönüyle tez kapsamında önerilen model-güdümlü otomatik dağıtım aracından ayrılmaktadır.

3.2. Mikroservislerin Dağıtımı için Önerilen Platform ve Araç Tabanlı Çalışmalar

Literatürde mikroservislerin dağıtımını ele alarak araç veya platform geliştiren çalışmalar incelendiğinde mikroservislerin iletişim parametreleri, hafıza kapasiteleri, düğümler üzerinde çalışma maliyeti gibi özellikler dikkate alınarak bir dağıtım modelinin geliştirilmediği görülmektedir. Önceki çalışmalarda [65] Graphical Modeling Framework ortamında paralel ve dağıtık simülasyon sistemleri için bir yaklaşım önerildiği ve bu yaklaşımı destekleyen araç [66] geliştirildiği görülmektedir. Ayrıca yazarlar bu yaklaşımı Veri Dağıtım Hizmeti (Data Distribution Service) tabanlı sistemler için de gerçekleştirmişlerdir [67].

Literatürde bu alanda önerilen diğer araçlar araştırıldığında algoritmik yaklaşımlardan ziyade dağıtımı yapacak özel bileşenlerin ve altyapıların kullanıldığı görülmektedir. Gabbrielli vd. [68], mikroservislerin bulut üzerinde dağıtık sistemlerin geliştirilmesi için uygun bir mimari yapıya sahip olduğundan, fakat bu servislerin dinamik yapısından dolayı otomatik yerleştirmeleri için uygun yöntemler gerektiğinden bahsederek mikroservislerin otomatik ve optimize edilmiş yerleştirimi için Jolie dilinde yazılan JRO isimli bir araç önermişlerdir. Geliştirilen araç Zephyrus, Jolie Enterprise (JE) ve Jolie Reconfiguration Coordinator (JRC) adlı üç bileşenden oluşmaktadır. Zeyhrus, hedef uygulamanın kısmi ve soyut tanımından başlayarak, kaç bileşenin gerekli olduğunu ve bunların sanal makinelere nasıl dağıtılacağını ve nasıl birlikte bağlanacaklarını gösteren mimarinin tüm detaylarını üreten bir araçtır. Jolie Enterprise, Jolie dilinde yazılan mikroservisleri yönetmek ve dağıtmak için geliştirilen dağıtık bir çerçevedir. Yönetilen sistemde çalışan servisler ve platformlar ile ilişkili tüm verilere erişmek, servisleri dağıtmak, başlatmak, bitirmek ve kaldırmak, aynı zamanda kaynak tüketimlerini ve performanslarını izlemek için bir API sergilemektedir. Son olarak JRC ise JE tarafından sağlanan bir dağıtım için istenilen bir yapılandırma ve içeriği verilen bir araçtır ve bu araç optimize edilmiş dağıtım planlamasını üretmek için Zephyrus ile etkileşimde bulunmaktadır. Yazarlar çalışmalarında mikroservis tabanlı bir uygulamayı optimal bir şekilde dağıtmaya yarayan JRO aracının literatürde ilk olduğunu savunmuşlardır.

Ciuffoletti [69] yaptığı çalışmada, izleme altyapısını uygulayan bir makine ile mikroservislerin otomatik olarak dağıtılmasını sağlamaktadır. Uygulamanın gerçeklemesi Docker çoklayıcı (hub) üzerinde yapılmış olup, çalışmanın katkısı, mikroservislere dayalı talebe bağlı bir izleme servisinin tasarımında altyapıyı tanımlayan modelden probları dağıtan makineye kadar tüm adımların tanımlanması olarak belirtilmiştir. Sonuçların Docker deposunda bulunan kaynaklar kullanılarak yeniden üretilebileceği ve genişletilebileceği bir ortam hazırlanmıştır. İzleme yapılandırması için kullanılan referans mimaride ayrı konteynerlerde yer alan sensör ve toplayıcı (collector) adlı iki uygulama yer almaktadır. Bir sensör, sensör uygulamasını barındıran bir konteyner olarak uygulanır. Kümeleme (Aggregation) ve yayın için karışımlar,

sensör uygulamasına eklenir ve kanallar kullanılarak birbirine bağlanır. Toplayıcı ise bir konteyner tarafından barındırılan bir sunucu olarak çalıştırılmaktadır. Konteyner tarafından çalıştırılan kaynağın metriklerini ölçer ve bunları sensöre iletir. Oluşturulan şema OCCI izleme belgesinin yönergelerini izleyerek Açık Bulut Hesaplama Arayüzü'ne (Open Cloud COmputing Interface) dayanmaktadır. Önerilen sensör/toplayıcı şemasının dezavantajlarından biri, tek bir toplayıcının birden fazla sensörü besleyememesi, başka bir deyişle, bir kollektörün çıkışının birden fazla sensöre çoklu yayın yapamamasıdır. Bu durum bir kollektörün bir OCCI bağlantısı olması gerekliliğinden kaynaklanmaktadır. İkinci dezavantaj olarak şema periyodik ölçümlerle başa çıkmak amacıyla basit bir şekilde tasarlandığı için asenkron alarmları yönetmenin bu sistemle mümkün olmadığı vurgulanmıştır.

Berger vd. [70], sürücüsüz araçlarda kullandıkları mimarilerini mikroservislere dönüştürmek amacı ile hem yazılım geliştirme hem de yazılımın dağıtımı için konteynerli yazılım paradigmasını başarı ile uygulamışlardır. Çalışmada sürücüsüz araçlar için bir yıllık çalışma ile uygulanan konteynerli geliştirme ve dağıtımı ile ilgili yaşanılan deneyimler aktarılmıştır. Yaklaşımda, yaygın olarak kullanılan Docker ekosisteminden birkaç bileşen kullanılmıştır. Mikroservis mimarisinden elde edilen deneyimlere göre; yazılım bileşenlerinin hesaplama düğümleri arasında sorunsuz bir şekilde başlaması ve geçis yapması için, durdurma ve yeniden başlatma açısından durumsuz bir davranış sergilemesi gerektiği gözlenmiştir. Konteynerli yazılım geliştirme ve dağıtım sürecinde elde edilen deneyimler sonucunda konteynerli yazılım geliştirmenin, dağıtılabilirlik, otomasyon ve izlenebilirlik özelliklerinden dolayı monolitik yapılandırmaya göre otomotiv endüstrisinde daha başarılı olduğu görülmüştür. Yazılım katmanlarından seçilen katmanları test ederken, özel Docker kayıtlarına (registry) itilen veya çekilen delta Docker görüntülerini sabit bir şekilde oluşturmaları, web ortamında dağıtım prosedürünü hızlandırmada önemli bir yere sahip olduğu için, araç bilgisayarlarının bir parçası olarak özel bir Docker kaydına sahip olmanın gerekliliği vurgulanmıştır.

Yapılan bir başka çalışmada, son üç yıl içerisinde önemli bir stratejik ve teknik değişim geçiren MGDIS SA yazılım düzenleme şirketinin çekirdek iş yazılımını tek parçalı mimariden mikroservisler kullanarak Web tabanlı mimariye dönüştürme süreci ele alınmıştır. Gouigoux ve Tamzalit [71], bu dönüşüm sırasında Web yönelimli mimariye başarılı bir şekilde adapte

edilmiş geçiş için, "Tek parçalı bir mimarinin en uygun tanecikli mikroservis API'lerine nasıl bölüneceği, mikroservislerin en iyi dağıtımının ve en verimli orkestrasyonunun nasıl belirlenebileceği" olmak üzere üç önemli soru üzerinde durmuşlardır. Çalışmada tek parçalı yapıdan mikroservislere geçişte tanecikli yapı arttıkça kalite güvencesinin maliyetinin azaldığı, dağıtım maliyetinin ise herhangi bir otomasyon olmadığında doğrusal bir şekilde, otomasyon olduğu durumda da asimptotik bir şekilde arttığı gözlemlenmiştir. Elde edilen deneyimlere bakıldığında, web yönelimli mimariye geçiş ve uygun mikroservislere eğilimin, servislerin farklı içeriklerde yeniden kullanımını artırdığı, değiştirme kolaylığı sağladığı (monolitik bir uygulama QA tarafından birkaç hafta içinde doğrulanırken tek bir servis yaklaşık bir günde doğrulanmaktadır. QA için toplam süre kabaca aynıdır, ancak mikroservis mimarisi, zamana yayılmasını mümkün kılarak, pazara giriş için daha iyi bir zaman kazandırır) ve yeniden kullanım ve değişimden sonra performans artışı, MGDIS'in yeni mimarisindeki üretimde gözlenmiştir. Özellikle kaba ve karmaşık bir web yönteminde, ortalama yanıt süresi 11 000 ms iken (bu değer bir performans optimizasyon kampanyasından sonra kaydedilmiş ve birçok çabaya ragmen eski mimaride daha fazla azaltılamamıştır), mikroservis mimarilerinin kullanımı ile 70 ile 300 ms arasında yanıt süresine inmiştir.

Mikroservislerin dağıtım problemini ele alan bir diğer çalışmada [72], bulut bilişim katmanları arasında yer alan SaaS altyapısının geliştiricilerinin çoklu kiracılık ve çok sayıda kullanıcıdan dolayı ölçeklenebilirlik, mevcudiyet, artan test/geliştirme maliyetleri gibi zorlukların olduğu belirtilmektedir. Bundan dolayı güçlü ve genel bir dağıtım platformuna ihtiyacın gün geçtikçe artmakta olduğu vurgulanmış, SaaS geliştiricilerini dağıtımlardan ayırmak ve en iyi uygulamalar ile gerçek dünya uygulamaları arasındaki uçurumu kapatmak için BigVM adlı yeni bir platform önerilmiştir. BigVM, SaaS geliştiricilerine, çok katmanlı mikroservis tabanlı şekilde SaaS çözümlerinin oluşturulması, özelleştirilmesi ve dağıtılmasını sağlamak için mikroservis odaklı dağıtım setleri sağlamaktadır. BigVM, özellikle çoklu uygulamalarda yüksek oranda yeniden kullanılan ve standartlaştırılmış mikroservislerin belirlenmesine odaklanmaktadır. Bu platform; hata toleransını kullanabilme, kaynakları optimize etme ve sadece kaynak kullanımına bağlı olarak ölçekleme değil zamanlama kısıtı gibi işlevsel olmayan gereksinimlere dayalı olarak da ölçekleme yapabilmektedir. Docker konteynerler

kullanılarak yapılan deneyler, Docker konteynerlerinin CPU iş yükü ve dosya G/Ç işlemleri açısından istenen performansı elde edebildiğini göstermiştir.

Singh ve Peddoju [47], mikroservislerin sürekli entegrasyonu ve dağıtımı sırasında meydana gelen zorlukları analiz ederek bu zorlukların üstesinden gelmek için mikroservislerin dağıtımına ve sürekli entegrasyonuna yardım eden otomatik bir sistem önermişlerdir. Önerilen mikroservis mimarisini Docker konteynerler üzerinde dağıtmış ve sosyal ağ uygulaması kullanarak test etmişlerdir. Sonuçlar, yanıt zamanı, üretilen çıktı (throughput), dağıtım zamanı gibi parametreler açısından monolitik yaklaşımla karşılaştırılmıştır. Mikroservis dağıtım deneyleri için API Proxy, Configuration Storage, Build Server, Source Repository, Container Repository gibi tasarım bileşenleri ile birlikte HaProxy, Consul, Jenkins, GIT Repository ve Docker Registery araçları kullanılmıştır. Deneyler sonucunda, mikroservis yaklaşımı kullanılarak geliştirilen uygulama ve önerilen tasarım ile yapılan dağıtımın mikroservislerin dağıtımı ve sürekli entegrasyon için zamanı ve çabayı azalttığı görülmüştür. Aynı zamanda, düşük tepki süresi ve yüksek bit akışından dolayı, mikroservis tabanlı uygulamanın monolitik tasarımı geride bıraktığı görülmüştür.

Guo vd. [50], mikroservis mimarisine ve hafif (lightweight) konteyner teknolojisine dayalı CloudwareHub isimli yeni bir Cloudware (internet ortamında çalışan yazılım) PaaS platformu önermişlerdir. CloudwareHub, bulut üzerindeki yazılımın geliştirilmesi, test edilmesi, dağıtılması ve yürütülmesi için kullanıcılara ve geliştiricilere araçlar sunan bir platformdur. Yazarlar bu platformda tarayıcı tarafından kullanıcılara hizmet sağlayan geleneksel yazılımları herhangi bir değişiklik yapmadan doğrudan dağıtabilmektedir. Mikroservis mimarisini kullanarak bu platform, ölçeklenebilirlik, otomatik dağıtım, felaket kurtarma ve esnek konfigürasyon özelliklerine sahip olmuştur. Yapılan deneyler sonucunda çoğu durumda istemcinin kararsız bir ağa sahip olduğu ve ağ ortamına dinamik olarak uymasının önemli olduğu belirtilmiştir. Bu nedenle, veri sıkıştırma ve bant genişliğinin gecikmeyi azaltıcı etkenler olduğundan ve CloudwareHub için yapılan bir önbellek sisteminin kullanıcı deneyimini büyük ölçüde artıracağından bahsedilmiştir. Çalışmada servislerin dağıtımı için konteyner teknolojileri dışında herhangi bir dağıtım aracının kullanılmadığı, temel olarak kullanıcıların doğrudan bulut üzerinden servisleri kullanabileceği bir platform tasarlandığı görülmektedir.

Profeta vd. [73] tarafından yapılan çalışmada farklı senaryo ve alanlarda Büyük Veri Analitik (Big Data Analytics-BDA) uygulamalarının dağıtımı, yürütülmesi ve bileşimi için mikroservis tabanlı bir platform önerilmektedir. ALIDA isimli bu platform, hem BDA uygulama geliştiricilerinin hem de veri analistlerinin etkileşime girdikleri birleşik bir platform olarak tasarlanmıştır. Geliştiriciler, maruz kalan API ve/veya web kullanıcı arayüzü üzerinden yeni BDA uygulamaları kaydedebileceklerdir. Veri analistleri ise bir veya daha fazla kaynaktan gelen sonuçları manipüle edip görselleştirmek için bir pano (dashboard) kullanıcı arayüzü üzerinden iş akışlarını oluşturmak amacıyla, sağlanan BDA uygulamalarını kullanabileceklerdir. Ayrıca önerilen bu platformun BDA uygulama iş akışlarının performans metriklerine dayalı bir makine öğrenmesi modeli sentezlenerek optimal dağıtım yapılandırması sağladığı belirtilmektedir. Platformun yalnızca BDA uygulamalarına özel tasarlanması, mevcut durumda farklı tipte mikroservis tabanlı uygulamalarda kullanılamamasına neden olmaktadır.

[74] referanslı çalışmada konteyner orkestrasyon dili sunularak bir servisin diğer servislere bağımlılıklarını belirleyen yönergelerle donatılmış pipekit isimli bir araç geliştirilmektedir. Bu araç, her bir servis için paylaşılan depolama birimi kullanılarak servisler arasında veri alışverişi yapmak için bir iletişim katmanı sağlamaktadır. Mikroservis mimarilerinin dağıtımı için genellikle ilk yapılandırmada farklı servisler arasında veri alışverişinin belirlenmesi gerektiğinden pipekit ile geliştirilen depolama birimleri sayesinde servisler hazır olduğu zaman sisteme bildirilerek dağıtım yapılmaktadır. Çalışmada Docker Compose gibi Docker orkestrasyon araçlarının karmaşık senaryoları yeteri kadar desteklemediği, Docker prensibini kullanan pipekit iletişim katmanı aracının da Docker Compose'un genişletilmiş versiyonu olarak tasarlandığı ve böylece iletişim katmanının paylaşılan depolama birimi olarak uygulanmasının herhangi bir ön yapılandırma adımı gerektirmediği belirtilmektedir.

3.3. Mikroservislerin Verimli Dağıtımı için Algoritmik Yaklaşım Öneren Çalışmalar

Wan vd. [75], Docker'ın özellikleri, mikroservis tabanlı uygulama gereksinimleri ve bulut veri merkezlerindeki mevcut kaynakları inceleyerek uygulama dağıtım problemini formülize

etmişlerdir. Daha sonra konteyner yerleştirimi ve görev atamasını belirlemek için bir çerçeve ve algoritma önermişlerdir. Önerilen algoritma dağıtık ve artan bir şekilde çalışarak çerçeve altında çok büyük fiziksel kaynaklara ve çeşitli uygulamalara ölçeklenebilmektedir. Geliştirilen çerçevede, uygulama istekleri Yürütme Konteynerleri (Execution Containers-ECs) üzerinde mikroservisler olarak işlenmektedir. Uygulamalar için kaynak tahsisi ve kaynak yönetimi Mikroservis Denetleyicileri (Microservice Controllers-MCs) üzerinde gerçekleştirilmektedir. Hipervizör tabanlı VM yerleştirmenin aksine yürütme konteynerlerinin sayısı ve bunların fiziksel kaynaklara olan talepleri, yalnızca uygulamaların işyükü değil aynı zamanda veri merkezindeki mevcut kaynaklar tarafından da dinamik olarak belirlenmektedir. Bir uygulama dağıtılacağı zaman birden fiziksel makine üzerinde dağıtılmış bir dizi konteyner uygulamasına kaynaklar tahsis edilmektedir. Yazarlar önerdikleri çözümü Google Cluster'da deneyler yaparak Docker Swarm'da bulunan üç strateji ile karşılaştırmışlardır. Sonuç olarak önerilen çerçeve ve algoritmanın var olan tekniklere göre daha çok esneklik ve daha az maliyet sağladığı gözlemlenmiştir.

Leitner vd. [9], mikroservis tabanlı uygulamaların açık buluta dağıtımı sırasında hesaplama ve G/Ç maliyetleri dahil olmak üzere dağıtım maliyetlerini modellemek için çizge tabanlı bir yaklaşım sunmuşlardır. CostHat adı verilen model, geleneksel IaaS ve PaaS bulutlarına dağıtılan mikroservisler ve AWS Lambda gibi yeni bulut programlama paradigmalarını kullanan servisleri desteklemektedir. Bir ağ modeline dayalı olan CostHat, maliyet getiren kod değişikliklerini Entegre Geliştirme Ortamı (IDE)'nda uyaran araçları içermektedir. Bağımsız bir Python uygulaması kullanılarak oluşturulan CostHat modelindeki maliyet hesaplamasının standart donanımlar üzerindeki hesaplamalardan ucuz olduğu görülmüştür. Bu durum, geliştiricinin kod üzerinde çalışırken arka planda bir uygulamanın maliyetini sürekli olarak yeniden değerlendiren geliştirici araçları için modelin gerçek zamanlı olarak kullanılmasını sağlamaktadır.

Carvalho vd. [76] tarafından yapılan çalışmada mikroservis tabanlı uygulamalarda yazılım mimarının bulut hizmeti seçimi için çoklu sağlayıcı seçim yaklaşımı önerilmektedir. Diğer yaklaşımlardan farklı olarak her bir mikroservisi tek bir sağlayıcıda dağıtmak yerine bu yaklaşım, bir mikroservis için tek bir sağlayıcının çeşitli hizmetlerini ve her mikroservis için ayrı

sağlayıcıları seçebilmektedir. Bulut hizmetlerini sıralamak için çok ölçütlü karar verme yöntemi kullanılmakta ve seçim süreci aç gözlü algoritma yaklaşımını kullanan çoktan seçmeli sırt çantası problemi ile eşleştirilmektedir. Yazarlar önerilen yaklaşımın performansını değerlendirmek için Python tabanlı bir araç geliştirmekte ve araç, mikroservisleri barındıracak sağlayıcıların yanı sıra kullanılacak hizmetleri ve her sağlayıcının maliyetini bir JSON dosyası olarak kullanıcıya sunmaktadır. Yapılan çalışmada en uygun dağıtım modeli için özellikleri önceden belirlenmiş sunucular bulunmamakta, aç gözlü çoklu bulut sağlayıcı seçim mekanizmasına odaklanılarak kullanıcıya maliyet bilgilendirmesi yapılmaktadır. Bu bağlamda çalışma kapsamında geliştirilen aracın fiziksel kaynak modellemesi için bir ön bilgilendirme olarak kullanılabileceği öngörülmektedir.

Sampaio Jr. vd. [31] mikroservis tabanlı bir uygulamanın performans ve kaynak kullamının, uygulamayı oluşturan mikroservislerin dağıtımına bağlı olduğunu vurgulayarak bu amaçla geliştirilen Kubernetes, Docker Compose gibi araçların minimal yeteneklerinin olduğunu belirtmektedirler. Bu yüzden dağıtım alternatifi olarak mikroservislerin otomatik yerleşmesini yöneten REMaP adlı bir adaptasyon mekanizması önerilmektedir. Önerilen mekanizma mikroservislerin yerleştirilmesinde servislerin birbirleri ile ilişkilerini (iletişim maliyetlerini) ve kaynak kullanım geçmişlerini kullanarak çalışma zamanında servis yerleşimini otomatik olarak değiştirebilmektedir. Bu yer değiştirmeyi yapabilmek için MAPE-K [77] tabanlı bir uyarlama yöneticisi kullanılmaktadır. Önerilen bu mekanizma mikroservislerin çalışma esnasındaki dağıtımını ele almış olup tasarım aşamasında sistemin değerlendirilmesine olanak tanımamaktadır. Bu açıdan önerdiğimiz araçtan ayrılmaktadır. Mikroservislerin optimal ve otomatik dağıtımını ele alan bir diğer çalışmada [29] düğümler üzerindeki maliyeti minimuma indigermek için "mikroservisler ve bu servislerin düğümler üzerinde dağıtımını belirleyen kısıtlama dizisini üretmek, kurulacak bağlantıları içeren kısıtlamalar dizisini üretmek ve son olarak bu kısıtılara bağlı olarak ilgili dağıtım planını sentezlemek" olmak üzere üç aşamada çalışan bir algoritma sunulmaktadır. Bu kısıtlamalar sayesinde dağıtımın tüm maliyetini minimuma indirgeyen optimizasyon metrikleri belirlenmektedir. Yazarlar geliştirdikleri tekniği test etmek için ABS [78] dilinde gerçek dünyadaki bir mikroservis mimarisini

modelleyerek bir dizi dağıtım planı hesaplamaktadırlar. Bu öneri, amaç olarak önerilen yaklaşıma oldukça yakındır fakat çalışma kapsamında geliştirilen araç, kullanıcıya herhangi bir ek donanım gerektirmeden farklı algoritmalar kullanılarak ortaya çıkarılan dağıtım alternatifi seçimini tasarımcıya bırakmaktadır. Bu çalışmada önerilen araç, tek bir algoritmik yaklaşım kullanılarak dağıtım planı oluşturmakta ve modelleme yapmak için ABS dili hakkında bilgi sahibi olmayı gerektirmektedir. Çalışma kapsamında önerilen Micro-IDE mikroservis dağıtım aracı, CTAP problemini çözebilen tüm algoritmaları desteklenmekte ve tasarım aşamasında minimum toplam maliyeti elde ederek birden fazla verimli dağıtım alternatifi sunarak kullanıcıya oluşturulan mimariye en uygun dağıtım alternatifine karar verme fırsatı sunmaktadır.

İlişkili çalışmalar ile önerilen Micro-IDE aracının teknik özellikler açısından karşılaştırması Çizelge 3.1.'de verilmektedir.

Çizelge 3.1. Literatürde yer alan dağıtım yaklaşımları ve Micro-IDE aracının karşılaştırılması

İlişkili Çalışma	Araç veya Platform tabanlı	Verimli dağıtım gözetimi	Algoritmik yaklaşım	Model-güdümlü tasarım
Ciuffoletti [69] (2015)	✓	✗	✗	✗
Leitner vd. [9] (2016)	✗	✓	✓	✗
Guo vd. [50] (2016)	✓	✗	✗	✗
Profeta vd. [50] (2016)	✓	✗	✗	✗
Gabbrielli vd. [64] (2017)	✓	✓	✗	✗
Granchelli vd. [63] (2017)	✓	✗	✗	✓
Dullman ve Hoorn [64] (2017)	✓	✗	✗	✓
Zheng vd. [72] (2017)	✓	✗	✗	✗
Singh ve Peddoju [47] (2017)	✓	✗	✗	✗
Guzmán vd. [74] (2018)	✓	✗	✗	✗
Wan vd. [75] (2018)	✗	✓	✓	✗
Carvalho vd. [76] (2019)	✗	✗	✓	✗
Sampaio Jr. vd. [31] (2019)	✓	✓	✓	✗
Önerilen Micro-IDE aracı (2021)	✓	✓	✓	✓

Çizelge 3.1. değerlendirildiğinde verimli dağıtım gözetimi yapan mikroservis tabanlı çalışmaların oldukça az olduğu görülmekte ve mikroservislerin dağıtımını yazılım yaşam döngüsünün ilk aşaması olan tasarım sürecinde yapılmasını sağlayan model güdümlü mimarinin ise genellikle tercih edilmediği görülmektedir. Literatürdeki çalışmalarda mikroservislerin dağıtımı için önerilen araç veya platformlarda Docker Swarm, Kubernetes gibi populer orkestrasyon araçlarına ek olarak mekanizmalar geliştirilmektedir. Mikroservis tabanlı bir mimarinin

tasarım aşamasında verimli dağıtım gözetiminin yapılması, yazılım yaşam döngüsünün ileri aşamalarında karşılaşılacak problemlerin önceden belirlenebilmesini sağlamaktadır. Bu bakımdan, tasarım aşamasında mikroservisler arası iletişim maliyetini gözeterek birden fazla algoritma seçeneği ile çeşitli verimli dağıtım alternatifleri sunması, bununla birlikte manuel dağıtımı desteklemesi ve farklı algoritmaların da araca esnek bir şekilde adapte edilmesi özellikleri ile önerilen Micro-IDE yaklaşımının literatüre ve endüstriye somut bir katkı sağlayacağı düşünülmektedir.

4. DURUM ÇALIŞMASI TASARLAMA VE PLANLAMA SÜRECİ

Bu tez çalışması kapsamında önerilen yaklaşımı değerlendirmek adına gerçek hayatta kullanılan sistemlerden Taksi Çağırma Sistemi [5] ve Spotify müzik uygulaması [18] durum çalışması olarak ele alınmaktadır. Yazılım mühendisliği yaklaşımında bir durum çalışması, özellikle olay ile bağlamı arasındaki sınırların açık olmadığı durumlarda, olayları kendi bağlamları içinde incelemek için kullanılmaktadır [79]. Bu çalışmada ele alınan durum çalışmalarında da gerçek dünyada dinamik olarak çalışan mikroservislerin isimlerine ve sayılarına ulaşılamadığı için çalışmanın yazarı tarafından uygulama arayüzleri incelenerek sistemlerin metamodelleri sezgisel olarak tasarlanmaktadır. Durum çalışmaları için Robson [80] tarafından yapılan sınıflandırma içerisinde dört farklı araştırma metodolojisi (keşfedici-exploratory, tanımlayıcı-descriptive, açıklayıcı-explanaroty ve iyileştirme-improving) bulunmaktadır. Bu metodolojiler içerisinde, çalışma kapsamında oluşturulan durum çalışmaları, bir durumu veya olayı tasvir etmek amacı ile kullanıldığı için "tanımlayıcı" olarak sınıflandırılmaktadır. Bu durum çalışmaları tanımlanırken aşağıdaki araştırma sorularına (AS) cevap aranarak önerilen yaklaşımın ve araç desteğinin somut olarak katkısı değerlendirilmektedir.

- AS1: Önerilen Micro-IDE aracı hedeflenen amacı gerçekleştiriyor ve doğru çalışıyor mu?
- AS2: Tasarlanan durum çalışmaları geliştirilen aracın performansını değerlendirmede yeterli mi?
- AS3: Önerilen yaklaşım ve araç desteği durum çalışmaları açısından değerlendirildiğinde diğer dağıtım yaklaşımlarına göre avantaj sağlıyor mu?
- AS4: Durum çalışmaları için geliştirilen metamodeller için parametre tahmini nasıl yapıldı?
- AS5: Önerilen yaklaşım için yapılan değerlendirmelerde sınırlamalar ve geçerliliğe yönelik tehditler nelerdir?

Belirlenen araştırma sorularına cevap aramak için öncelikle Bölüm 4.1.'de yer alan örnek durum çalışmaları spesifik olarak araç üzerinde tasarlanarak Micro-IDE aracı üzerinde durum çalışmaları için oluşturulan Mikroservis Çalışma Zamanı Yürütme Modeli ve Mikroservis Altyapı Modeli, Mikroservis Dağıtım Modeli üretmek amacı ile kullanılmaktadır. Bu süreçte AS1 sorusunun yanıtını aramak için önerilen Micro-IDE aracı kullanılarak tasarlanan durum çalışmaları için hem manuel hem de algoritmik yaklaşımlar kullanılarak dağıtım alternatifleri üretilmektedir. Somut olarak birden fazla verimli dağıtım modeli elde edilip bu modellerin analizi ve değerlendirmesi detaylı bir şekilde yapılabildiği için önerilen bu aracın hedeflenen amacı gerçekleştirebildiği söylenebilmektedir. Algoritmik yaklaşımlar aracılığı ile dağıtım yapan bu araç üzerinde tanımlanan mikroservislerin toplam kapasitesi, atanacak olan düğümlerin toplam kapasitesinden büyük olduğu durumda Micro-IDE aracı geri bildirim vererek yeterli hafıza olmadığı konusunda tasarımcıyı uyarmaktadır. Bu durumda herhangi bir dağıtım alternatifi üretilmeden tasarımcı Çalışma Zamanı Yürütme Konfigürasyon Modeli'ne yönlendirilmektedir. Önerilen araçta her bir algoritmik yaklaşım farklı minimum maliyet hesabı yaparak farklı dağıtım alternatifleri türettiği için aracın doğrulama işlemi hesaplanan iletişim maliyetleri ve çalıştırma maliyetlerinin servis bazında incelenmesi ile yapılmaktadır. Aracın doğru çalıştığı, algoritmaya özgü dağıtım modeli analiz raporunda yer alan detaylı maliyet hesapları ve hafıza kapasiteleri ile kanıtlanmaktadır. Bir diğer araştırma sorusu olan AS2'ye cevap olarak, tasarlanan durum çalışmalarında karmaşık mikroservis iletişim desenleri, mikroservislerin dağıtımı için gerekli temel parametreler irdelendiğinden dolayı geliştirilen aracın performansını değerlendirmede bu çalışmaların yeterli olduğu söylenebilmektedir. AS3'e cevap olarak çalışmanın değerlendirme bölümünde (Bölüm 9.) manuel ve algoritmik yaklaşımlar birbirlerine göre karşılaştırılarak önerilen yaklaşımın diğer mevcut yaklaşımlara göre avantajları detaylandırılmaktadır. Ayrıca Bölüm 3.'de yer alan literatür çalışmalarında mikroservislerin dağıtımı için önerilen yaklaşımların çalışma kapsamında önerilen yaklaşıma göre farklılıkları somut bir şekilde değerlendirilerek Micro-IDE aracının diğer yaklaşımlara göre avantajları belirtilmektedir. Geliştirilen metamodeller arasından tahmini parametrelerin belirlenmesi işlemi sistemin çalışma zamanı verilerini içeren Mikroservis Çalışma Zamanı Yürütme Modeli üzerinde yapılmaktadır. Bölüm 6.5.1.'de AS4'e

cevap olarak gerekli bilgiler verilmektedir. Son araştırma sorusu (AS5) için tespit edilen sınırlamalar ve geçerliliğe yönelik tehditler Bölüm 10.1.'de detaylandırılmaktadır.

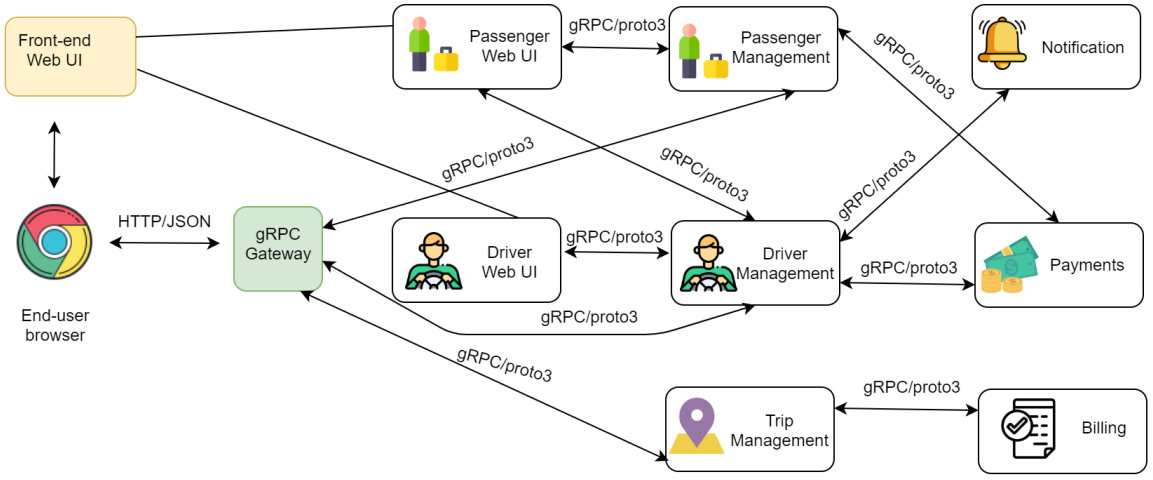
4.1. Örnek Durum Çalışmaları

4.1.1. Taksi Çağırma Sistemi-TÇS

Bu bölümde problem tanımını örneklendirmek ve önerilen yaklaşımı değerlendirmek için bir durum çalışması benimsenmektedir. Bu bağlamda mikroservis mimarilerine geçiş yaparak neredeyse bir çok ülkede kullanılır hale gelen Uber [5]' in taksi çağırma uygulaması, durum çalışması olarak belirlenmektedir. Birçok servis sisteminde olduğu gibi Uber de başlangıçta tek bir şehir için geliştirilerek yekpare bir yapıda başlangıç yapmaktadır [26]. O zamanlar tek bir şehirde kullanıldığı için tek bir kod tabanı, Uber'in temel iş (business) problemlerini çözebilmekte idi. Tek parçalı TÇS, yolcular ve sürücüler arasındaki bağlantıyı sağlayan bir REST API'ye sahiptir. Bu API ile birlikte kullanılan üç farklı adaptör de bir taksi rezervasyonu yaparken faturalama (billing), ödemeler (payments), e-posta/mesaj gönderme gibi işlemlerin gerçekleşmesini sağlamaktadır. Ayrıca sistemin tüm verileri tek bir veritabanında saklanmaktadır. Böylece yolcu yönetimi (passenger management), sürücü yönetimi (driver management), bildirim (notification), ödemeler (payments) ve yolculuk yönetimi (trip management) gibi tüm özellikler tek bir çerçevede oluşturulmaktadır.

Uber'in taksi çağırma sistemi (taxi-hailing system) dünya çapında genişlemeye başladıkça tek bir kod tabanı bir çok karmaşıklığa yol açmış ve bu da ölçeklenebilirlik ve sürekli entegrasyona (Continuous Integration-CI) bağlı problemleri beraberinde getirmiştir [26]. Örneğin tek bir özelliği güncellemek için tüm özelliklerin defalarca yeniden inşa edilmesi, test edilmesi ve dağıtılması gerekmektedir. Tek bir depoda hataların kaynağını bulup düzeltmek bir geliştirici için oldukça zor bir hale gelmektedir. Ayrıca dünya çapında yeni özelliklere göre sistemin var olan özellikleri eşzamanlı güncellendikten sonra anlık olarak bu özellikleri ölçeklemenin maliyeti tek parçalı mimarilerde oldukça yüksektir. Geliştirme problemlerinin yanı sıra tek parçalı uygulamaların çalıştırılması da oldukça zordur. Örneğin, bir özelliğin talep üzerine ölçeklendirilmesi, tüm özelliklerin birlikte ölçeklenmesini gerektirir ve bu da

tek parçalı mimarilerde ölçeklenebilirlik maliyetini artırmaktadır. Bu problemlerden dolayı Uber de Netflix [3], Amazon [4], Twitter [81] ve daha birçok populer şirketi takip ederek mikroservis mimarilerini benimsemektedir. Şekil 4.1.'de mikroservis mimarisi ile tasarlanan Uber'e ait bir taksi çağırma sistemi gösterilmektedir [5]. Mimaride Billing, Payments, Passenger Management, Driver Management, Notification, Trip management, Passenger Web UI ve Driver Web UI olmak üzere 8 farklı mikroservis bir API Gateway aracılığı ile birbirlerine bağlanmaktadır. Her bir servis birbirlerinden olabildiğince izole edilmektedir. Böylece birimler ayrı işlevleri gerçekleştiren ayrı ayrı konuşlandırılabilir üniteler haline gelmektedir. Yani bir servisteki değişiklik sadece o servisin dağıtımını gerektirmekte ve her servis bireysel olarak ölçeklendirilebilmektedir.



Şekil 4.1. Taksi çağırma uygulamasının mikroservis mimarisi

Servis örneği sayısı, bu servislere yapılan anlık talep doğrultusunda değişmektedir. Örneğin taksi arayan kişi sayısı taksi rezervasyonu yapan ve ödeme yapan müşteri sayısından daha fazla olduğu için farklı servislere ait mikroservis örnekleri sayısı da talebe göre değişmektedir. Bu durum yolcu yönetim (passenger management) örneklerinin ödemeler (payments) örneklerine göre sayıca fazla olmasını gerektirmektedir. Ayrıca servisler arasındaki iletişim sıklığı anlık isteklere göre değişmekte ve servisler arasında farklı iletişim türleri kullanılabilmektedir. Uygulamayı kullanan bir kullanıcı yolculuk talebinde bulunduğunda aşağıdaki adımlar tetiklenmektedir [82].

- Trip Management servisi API Gateway tarafından uyarılmaktadır.
- Trip Management servisi, istek/yanıt eş zamanlı iletişim türü ile Passenger Management servisinden yolcu bilgilerini istemektedir.
- Bu bilgileri aldıktan sonra Trip Management, Sevk Görevlisi'ne (Dispatcher) yolculuk ayrıntılarını bildirmektedir.
- Ardından, Dispatcher uygun bir sürücüyü buluarak hem Passenger Management hem de Driver Management servislerini aynı anda yayınla/abone ol iletişim kanalı aracılığıyla bilgilendirmektedir.

Bir durum çalışması olarak yukarıda tanımlanan sistem, önerilen yaklaşımı doğrulamak için kullanılmaktadır. Çizelge 4.1.'de görüldüğü gibi mikroservis isimleri ve ve bu servislerden kaç adet örnek kullanıldığı belirlenmiştir. Bunun yanısıra düğümler ve mikroservislerin hafıza gereksinimleri, düğümler üzerinde mikroservis örneklerinin çalışma maliyetleri ve bu servislerin birbirleri ile iletişim desenleri ve iletişim sıklıkları da tasarımcı tarafından tanımlanmaktadır.

Çizelge 4.1. TÇS için mikroservis örnek sayılarını içeren senaryo

Mikroservis adı	Örnek sayısı
Billing	50
Payment	15
Notification	125
Driver Web UI	30
Passenger Web UI	25
Driver Management	100
Passenger Management	150
Trip Management	55
Toplam	**550**

4.1.2. Spotify Müzik ve Podcast Uygulaması

İkinci durum çalışması olarak milyonlarca insanın aynı anda kullanabildiği Spotify uygulamasının bir bölümünden oluşan mikroservis yapısı, geliştirilen araç üzerinde tasarlanmaktadır. Gerçek dünya uygulamasında kullanıcılara kesintisiz hizmet vermek için yüzlerce mikroservis örneğinden oluşan Spotify uygulamasının yalnızca bir modülünde yaklaşık 18 adet mikroservis tespit edilmiştir. Mimarinin mantıksal altyapısı GOTO 2015 konferansında Kevin Goldsmith tarafından gerçekleştirilen sunumda yer almaktadır [83].

Çizelge 4.2. Spotify uygulaması için mikroservis örnek sayılarını içeren senaryo

Mikroservis adı	Örnek sayısı	Hafıza miktarı (MB)
Delivery Ingestor	50	20
Transcoder Control	58	25
Transcoder Service	68	15
Ingester	66	10
Merger	64	10
Non-Music Metadata Pipeline	78	30
Vmd2-cms	98	40
Content API	96	20
Scatman	76	25
Indexer	78	25
CurationUI	74	25
Preludex	77	20
NERD	66	30
ENsnapshot	87	30
Google docs	84	45
Group Curation	93	35
Splash Content	92	50
Vmd2-service	89	40
Toplam	**1361**	**40820**

Spotify uygulaması önerilen yaklaşımı değerlendirmede kullanmak için mikroservis örneklerinin tanımlandığı, mikroservislerin veri alışverişleri, servislerin her bir düğüm üzerindeki çalışma maliyetleri, düğümlerin hafıza kapasiteleri ve çalışma maliyetleri olmak üzere bir

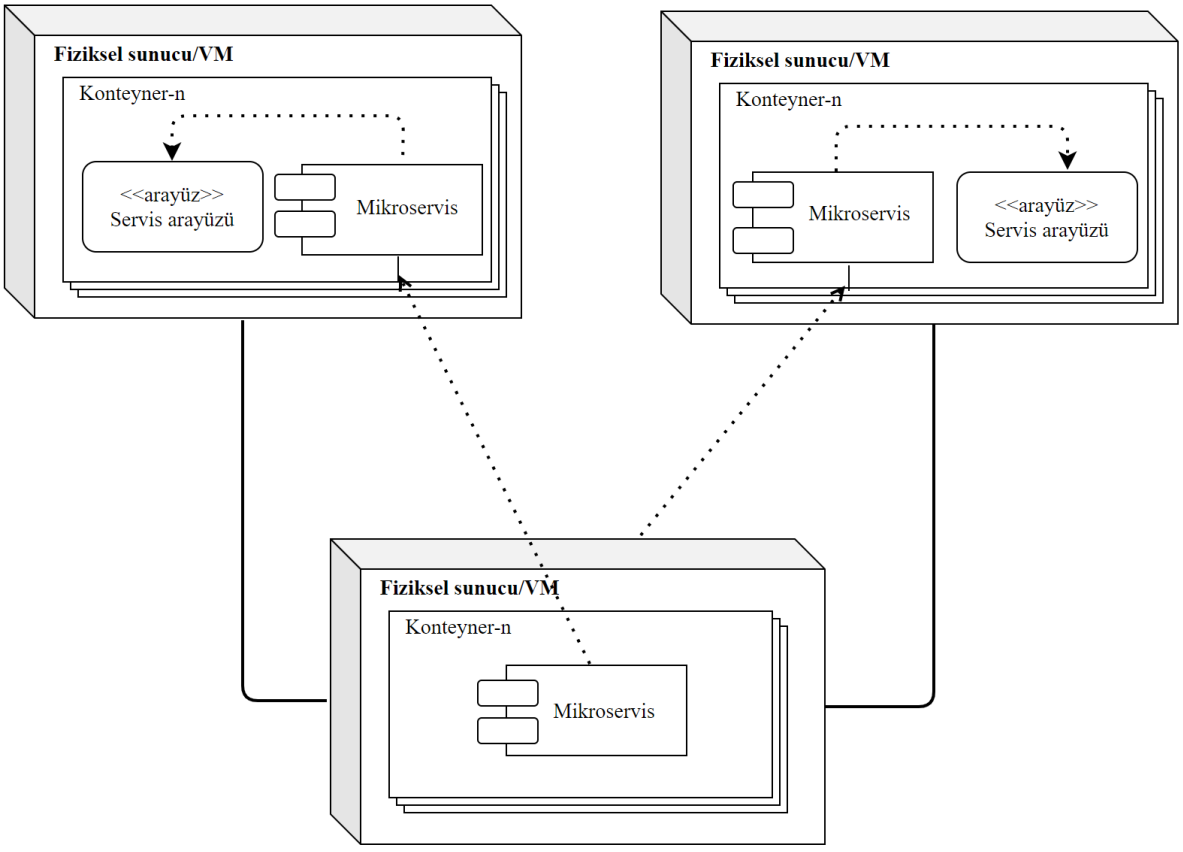
çok parametre belirlenerek örnek bir benzetim ortamı oluşturulmaktadır. Çizelge 4.2.'de örnek benzetim ortamında oluşturulan her bir mikroservisten kaç adet tanımlandığı gösterilmektedir.

Çizelge 4.2.'de verilen mikroservis örneği adı, Spotify uygulamasında yer alan mikroservisleri temsil etmektedir. Örnek sayısı ise verilen senaryoda bir mikroservisten kaç adet tanımlandığını göstermektedir. Son sütunda ise bir mikroservisin depolanması için gerekli hafıza megabayt cinsinden sunulmaktadır. Çizelge 4.2.'de tanımlanan değerlere göre toplamda 1361 mikroservis örneği, 40820 MB yer kaplamaktadır. Verilen senaryoda görüldüğü gibi mikroservis örnekleri sayısının oldukça fazla olduğu bir sistemde verimli dağıtım yapılması, toplam iletişim ve çalıştırma maliyeti açısından minimuma yakın bir sonuç elde etmeye bağlıdır. Binlerce servisten oluşan sistemin manuel olarak dağıtılması ile minimuma yakın bir maliyete sahip dağıtım modeli elde etmek oldukça zorlu bir süreçtir.Bu yüzden sistemin dağıtım performansını tasarım aşamasında algoritmik olarak değerlendirmek için model güdümlü Micro-IDE aracı önerilmekte ve Bölüm 6.' de bu aracın geliştirilmesi için oluşturulan metamodeller açıklanmaktadır.

5. PROBLEM TANIMI ve ÖNERİLEN YAKLAŞIM

5.1. Mikroservisler için Referans Mimari

Önerilen yaklaşım için dağıtım modeli olarak literatürde yer alan mikroservis dağıtım modelleri arasından "her konteyner başına bir servis örneği" modeline odaklanılmaktadır. Bunun sebebi "her VM başına bir/çoklu servis örneği" modelinden daha esnek ve hızlı olması ve sunucusuz mimarinin Bölüm 2.2.'de anlatıldığı gibi tüm servis örneklerine uyarlanabilir olmamasıdır. Şekil 5.1. "her konteyner başına bir servis örneği" için referans mimariyi göstermektedir. Önerilen yaklaşım "her konteyner başına bir servis örneği"'ne odaklanmasına rağmen "her VM için bir servis örneği"'ne de uygulanabilmektedir.



Şekil 5.1. Mikroservis referans mimarisi

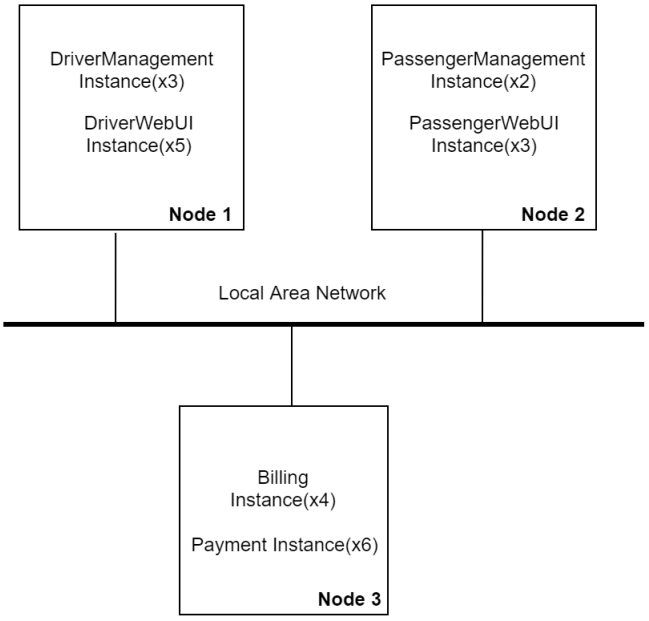
Mikroservis örnekleri, RESTful [84] hizmetleri, gRPC [85], yayınlama/abone olma (publish/subscribe) iletişim kanalları (örneğin; OMG DDS [55], Amazon SNS [86]) ve mesaj kuyruğu protokolleri (örneğin; Apache Kafka [38], Amazon SQS [87], RabbitMQ [37], ActiveMQ [39], Java Messaging Service [88]) gibi farklı protokollere dayalı olabilmektedirler. Bu örnekler, servis arayüzleri aracılığıyla farklı protokoller üzerinden birbirleriyle iletişim kurarak veri alışverişi yapabilmektedirler.

Ayrıca ihtiyaçlara göre servisler arasında hibrit iletişim modeli adı verilen hem eş zamanlı hem de eş zamansız iletişim modelleri kullanılabilmektedir [33]. Bahsedilen hibrit iletişim mimarisi, SOA'ya kıyasla mikroservis yaklaşımının en büyük avantajlarından biridir. HTTP, gRPC ve REST gibi eşzamanlı iletişim yöntemlerini kullanarak, bir servis başka bir servise bir istek göndererek ilgili servis yanıt verene kadar beklemektedir. Verilere hemen ihtiyaç duyulduğunda, eş zamanlı çalışan istek/yanıt iletişim modelleri uygundur. Fakat servisin uygun olmadığı veya anlık cevap veremediği durumlar olabilmektedir. Bu durumda, ölçeklenebilirlik sorunlarına neden olabilecek gecikmeleri önlemek için, bir mesaj kuyruğu veya yayınlama/abone olma modeli kullanılarak servisler arasında eş zamansız iletişim tercih edilebilir. Böylelikle, bir mikroservis mimarisinde aynı sistem içerisinde farklı iletişim protokolleri benimsenerek hem eş zamanlı hem de eş zamansız iletişim modelleri kullanılabilmektedir.

5.2. Problem Tanımı

Bulut ortamında mikroservis tabanlı bir uygulamanın performansını etkileyen anahtar sorunlardan biri mikroservislerin mevcut kaynaklara tahsis edilmesidir. Sınırlı sayıda servis ve düğüm ile küçük veya orta ölçekli uygulamalar için servis dağıtımı sistemi iyi bilen bir uzman tarafından etkili bir şekilde gerçekleştirilebilmektedir. Fakat mikroservis mimarilerine geçiş yapan çoğu yazılım sistemi genellikle büyük ölçekli ve çok sayıda servis ve düğümden oluşmaktadır. Büyük bir yazılım sisteminin bir uzman tarafından manuel olarak dağıtılması, verimli yerleştirme sürecini zorlaştırmaktadır. Ayrıca sistem büyüdükçe dağıtım süreci için minimum maliyeti bulmak imkansız bir hale gelmektedir. Düğümler ve servislerin hafıza kapasiteleri, servisler arası iletişim maliyetleri ve servislerin düğümler üzerindeki çalışma

maliyetleri gibi bir çok parametre düşünüldüğünde örnek bir senaryo için çeşitli alternatiflerin türetilebileceği görülmektedir. Örneğin taksi çağırma sistemi için billing, payment, passenger ve driver örnekleri üzerinden örnek bir dağıtım alternatifi Şekil 5.2.'de görüldüğü gibi tanımlanabilmektedir.

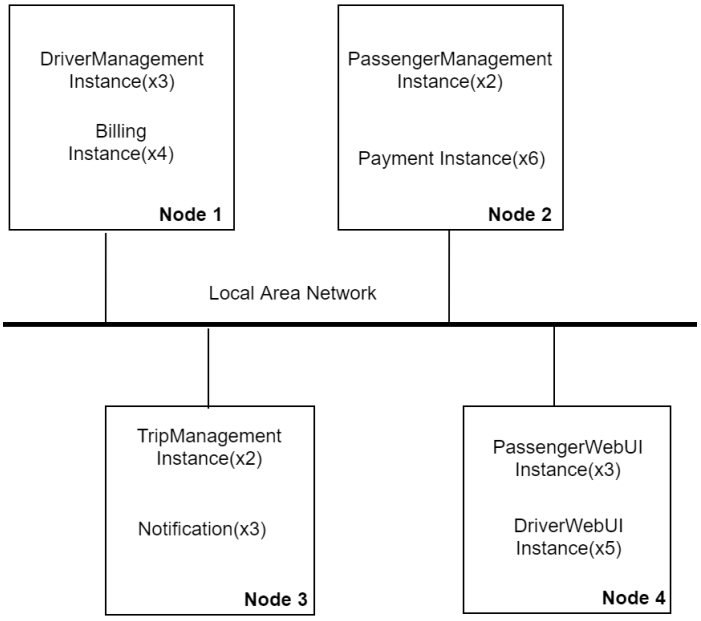


Şekil 5.2. Aynı servis örneklerinin bir grupta toplandığı örnek bir dağıtım alternatifi

Şekil 5.2.'de toplam 23 örnekten oluşan 6 farklı türde mikroservis 3 farklı düğüme dağıtılmaktadır. Bu durumda tek bir düğüme dağıtılan aynı türden örnekler arasındaki iletişim maliyeti sıfır olarak varsayılmaktadır. Eğer farklı düğümlere dağıtılan iki mikroservis örneği arasında yüksek sıklıkta iletişim var ise (örneğin, driver ve passenger servis örnekleri), yapılan dağıtım performans açısından değerlendirilmeli ve farklı alternatifler ile karşılaştırılmalıdır.

İkinci örnek dağıtım alternatifi Şekil 5.3.'de görülmektedir. Burada 8 farklı tür mikroservisten oluşan toplam 28 adet servis örneği 4 düğüm arasında eşit olarak dağıtılmaktadır. Yani iki farklı mikroservis türü her bir düğüme dağıtılmaktadır. Bu dağıtım yaklaşımı basit ve anlaşılması kolay olmasına rağmen aynı düğüme dağıtılan iki servis birbirleri ile sık iletişimde olmayabilir veya farklı düğümlere atanan servislerin birbirleri ile sık haberleşme ihtiyacı olabilir. Diğer bir problem ise kaynakların etkin kullanımıdır. Örneğin, düşük belleğe sahip

bir düğümün tüm hafızasını kullanmak veya yüksek kapasiteli bir düğümün yetersiz kullanımı, sistem performansını ve işletim maliyetini olumsuz etkileyebilmektedir. Bu nedenle, bu yerleştirme alternatifi, toplam iletişim maliyetinin en aza indirilmesi ve bellek kapasitesinin verimli kullanılması açısından uygun olmayabilir.



Şekil 5.3. Servislerin ikili olarak dağıtıldığı ikinci dağıtım alternatifi

Şekil 5.2. ve Şekil 5.3.'deki dağıtım alternatifleri gibi, çeşitli birçok dağıtım alternatifi mevcut düğüm sayısı, dağıtılacak servis sayısı, servisler arasındaki iletişim maliyetleri, servislerin her bir düğüm üzerindeki çalışma maliyetleri gibi parametreler göz önüne alınarak türetilebilir. Türetilen alternatifler arasında en iyiye en yakın performansı manuel olarak bulmak oldukça zordur. Ayrıca her bir dağıtım alternatifi, anlaşılabilirlik için mantıksal ayırma, ek yükü en iyi duruma getirme, fiziksel kaynakların kullanımını arttırma vb. gibi farklı kalite hususları açısından farklı performanslar göstermektedir. Bu yüzden uzman tarafından belirlenmesi çok zor olan verimli dağıtım alternatiflerinin otomatik olarak üretilmesi için daha sistematik ve formal bir yaklaşıma ihtiyaç duyulmaktadır. Ek olarak bu alternatiflerin minimum maliyet açısından birbirleri ile karşılaştırılması en verimli dağıtım alternatifinin belirlenmesi açısından önem arz etmektedir. Bir dağıtım yaklaşımı olarak konteyner teknolojileri

endüstride mikroservis uygulamalarının dağıtımı için sıklıkla kullanılmaktadır. Fakat bu teknolojiler dağıtım yapılandırmasını tanımlamak için konfigürasyon dosyalarına ihtiyaç duymaktadır. Yazılım yaşam döngüsünün tasarım aşamasında performansı optimize etmek için mikroservislerin dağıtımına ve kaynak tahsisine rehberlik edecek net bir yaklaşım yoktur. Bununla birlikte literatürde, dağıtım alternatiflerinin seçimi için nitelikli bir yaklaşım veya araç desteğinin önerilmediği görülmektedir. Devam eden bölümlerde mikroservis tabanlı uygulamalar için verimli dağıtım alternatiflerini algoritmik olarak türetebilen bir yaklaşım ve bu yaklaşımı destekleyen araç ailesi açıklanmaktadır.

5.3. Otomatik Verimli Dağıtım Alternatifleri Üretmek için Önerilen Yaklaşım

Bu bölümde mikroservis tabanlı sistemler için verimli yerleştirme seçeneklerinin tanımlanması ve değerlendirilmesi için önerilen somut bir yaklaşım sunulmaktadır. Sunulan yaklaşım, sistemin kodlanması ve geliştirilmesi tamamlanmadan önce tasarım aşamasında kullanılabilmektedir. Yaklaşımın tasarım aşamasında kullanılmasının temel avantajı verilen gereksinimleri en iyi şekilde karşılayarak etkin bir gerçekleştirim sağlamaktır. Yerleştirme tasarımının değerlendirilmesi ve başarımın analiz edilmesi tasarım aşamasında yapıldığı zaman detaylı tasarım, uygulama, test ve çalıştırma gibi geliştirme faaliyetlerinin gereksiz yere tekrar edilmesi önlenerek işgücü kaybı azaltılmaktadır. Böylece daha az işgücü kaybı ile daha az maliyetli ürün geliştirme olanağı sunulmaktadır.

5.3.1. Otomatik Mikroservis Dağıtımı Türetmek için Süreç Adımları

Bu çalışmada, mikroservislerin otomatik olarak dağıtılması için gerekli süreç adımları aşağıdaki gibi ilerlemektedir:

1. *Mikroservis Veri Değişim Modeli tasarımı:* İlk adım olarak mikroservis mimarisini oluşturan ekip tarafından iletişim esnasında mikroservisler arasındaki değişim yapılan verilerin tipleri ve nesne boyutlarını tanımlayan bir model oluşturulmaktadır.

2. *Sistemin mikroservislere ayrıştırılması:* İkinci adımda ekip, sistemi mikroservislere ayırarak bu servisler için gereksinimleri oluşturmaktadır. Tüm servislerin yeni servisler olması gerekmediği unutulmamalıdır. Genel olarak, ekibin yeni mimariye taşınabilen bazı eski servisleri vardır. Ekibin, uygulanabilir dağıtım oluşturmayı sağlamak için tasarım aracında hem yeni hem de eski mikroservisleri tanımlaması gerekir.

3. *İletişim deseni tasarımı:* Bu adımda ekip, ikinci adımda tanımlanan mikroservisler arasındaki iletişim modeline ve birinci adımda tasarlanan mikroservisler arasındaki veri alışverişi nesnelerine karar verir. Daha önce bahsedildiği gibi, tüm mikroservislerin veri alışverişi için aynı iletişim protokolünü kullanma zorunluluğu yoktur, ancak iletişim maliyetini tek tip bir şekilde değerlendirmek için bir süper set veri modeli tanımlanmaktadır. Çalışmanın ilerleyen bölümlerinde bu tek tip veri modeli hakkında daha fazla ayrıntı verilmektedir.

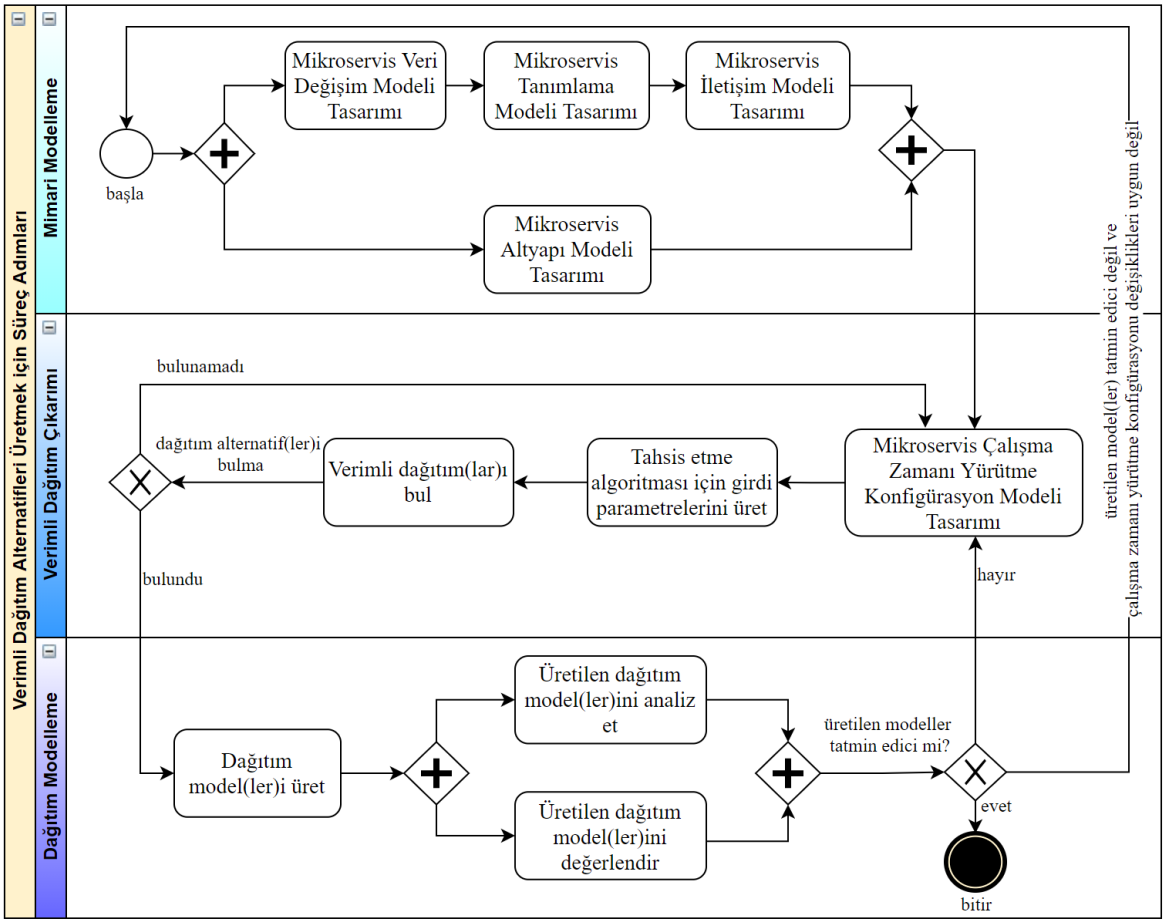
4. *Fiziksel kaynakların tanımlanması:* Bu adımda ekip, tasarım aracındaki mevcut kaynakları tanımlamaktadır. Fiziksel/sanal sunucular, CPU, bellek gücü ve aralarında bir ağ bağlantısı ile tanımlanmaktadır.

5. *Örnek çalıştırma senaryolarının tasarlanması:* Bu adımda ekip, ilk iki adımda tanımlanan mikroservisler ve veri alışverişi yöntemlerine/nesnelerine göre çalışma zamanı senaryolarını tanımlamaktadır. Çalışma zamanı senaryoları, her mikroservis örneği sayısını ve servisler arasında her iletişim kanalı için veri güncelleme/getirme (update/-fetch) oranlarını tanımlamaktadır. Ayrıca mikroservis örneklerinin mevcut kaynaklar üzerindeki yürütme maliyetleri tanımlanmaktadır.

6. *Verimli dağıtım alternatiflerinin türetimi:* Mikroservislerin tasarımından sonra veri alışverişi nesneleri, iletişim protokolleri ve fiziksel altyapısını kullanarak geliştirilen araç, dağıtım modeli oluşturma algoritması parametrelerini tasarımdan otomatik olarak çıkarmaktadır. Araç, CTAP (Kapasite Kısıtlı Görev Atama Problemi) çözücü algoritmasına uyarlanmış optimizasyon yöntemlerini çalıştırarak verimli dağıtım alternatifleri türetmektedir.

7. *Üretilen modellerin birbirleri ile karşılaştırılması ve sonuçların analiz edilmesi:* Bu adımda, araç, oluşturulan dağıtım alternatifi için yüksek trafik yükü oluşturan servisler, dağıtılan servis örneklerinin düğümler üzerindeki toplam hafıza kapasiteleri gibi bilgileri listeleyen bir karşılaştırma raporu oluşturmaktadır. Geliştirilen araç, tasarımcıya aynı algoritma üzerinden birden fazla çalıştırma ile oluşturulan iki farklı modeli ve farklı algoritmalarla veya manuel olarak oluşturulan iki dağıtım alternatifini karşılaştırma olanağı sunmaktadır.

Önerilen yaklaşımın İş Süreçleri Modelleme Notasyonu (Business Process Model and Notation-BPMN) diyagramı Şekil 5.4.'de gösterilmektedir. Yaklaşımda öncelikle veri alışverişinde kullanılacak olan Mikroservis Veri Değişim Modeli tasarlanmaktadır. Bu model ile kullanılan veri nesneleri ve nesne boyutları tanımlanmaktadır. Ardından mimaride kullanılacak mikroservisler tasarlanmakta ve Mikroservis Veri Değişim Modeli ile entegre çalışan ve servislerin hangi verileri ortak kullanacağını içeren iletişim protokolleri (Mikroservis İletişim Modeli) tanımlanmaktadır. Bununla birlikte mikroservislerin birbirleri ile haberleşeceği kanallar belirlenerek bu kanallar modellenmektedir. Bu modellerin yanısıra servisleri barındıran Mikroservis Altyapı Modeli, Mikroservis Veri Değişim Modeli, Mikroservis Tanımlama Modeli ve Mikroservis İletişim Modeli'nden bağımsız olarak oluşturulabilmektedir.



Şekil 5.4. Önerilen yaklaşımın BPMN diyagramı

Sistem mimarisinin tasarım aşaması tamamlandığında verimli dağıtım alternatifi üretme süreci Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli (Microservice Runtime Execution Configuration Model) ile başlamaktadır. Bu model, her mikroservis örneği sayısını ve tasarım aşamasında tanımlanan yapıları kullanarak her bir iletişim için güncelleme sıklığını (update rate) tanımlamaktadır. Tüm modeller geliştirildikten sonra bu modellerden girdi parametreleri çıkarılarak verimli dağıtım alternatifi üretme sürecine geçilmektedir. Eğer sistem girilen parametrelere göre uygun dağıtım alternatifi bulamazsa, modelleri güncelleme için sürecin başlangıç adımına tasarımcıyı yönlendirmektedir. Tasarımcı, Mikroservis Veri Değişim Modeli'nde tanımlanan veri nesnelerini değiştirebilme, ayırabilme veya birleştirebilme; mikroservisler arasındaki gereksiz iletişim kanallarını kaldırabilmekte veya daha iyi uyum ve düşük bağlantı için mikroservisleri birleştirebilme/bölebilme yetkinliğine sahiptir.

Eğer sistem bir veya daha fazla dağıtım alternatifi bulursa, dağıtım modeli oluşturma algoritmasının çıktısına göre üretilen dağıtım alternatiflerini tasarımcıya sunmaktadır.

Sürecin yedinci adımı önerilen yaklaşımın dağıtım modelleme bölümünü temsil etmektedir ve bu bölümde üretilen dağıtım modellerinin analizi ve karşılaştırılması yapılmaktadır. Bu bölümde sistem her bir iletişim kanalı için değiş tokuş edilen veri miktarı ve en fazla iletişimde olan servisler gibi bilgileri içeren bir dağıtım modeli analiz raporu oluşturmaktadır. Sistem aynı zamanda iletişim ve yürütme maliyetleri açısından farklı dağıtım alternatiflerinin otomatik olarak karşılaştırılmasına olanak tanımaktadır. Böylece sistem, tasarımcıların oluşturulan dağıtım modellerini karşılaştırma raporlarıyla analiz etmesine olanak tanımaktadır. Raporların sonuçlarına göre, üretilen modelin kabul edilebilir olup olmadığına tasarımcı karar vermektedir. Tasarımcı, iletişim ve yürütme maliyetlerini daha da iyileştirmek için tasarımın veya fiziksel kaynakların güncellenmesi konusunda hala sorunlar olduğunu düşünürse, tasarım aşamasının başına dönebilmekte ve süreci tekrarlayabilmektedir.

Genel bir perspektif açısından bakıldığında bir çok bileşenden oluşan bir sistem için verimli yerleştirme alternatifi bulma, matematiğin optimizasyon dalında yer alan polinomsal karmaşıklıkta bir problemidir ve literatürde yer alan kapasite kısıtlı görev atama problemi (Capaciatated Task Assignment Problem-CTAP) ile birebir örtüşmektedir. Görev atama probleminin uygulanması için öncelikle girdi parametrelerinin açık bir şekilde belirlenmesi gerekmektedir. Bu girdi parametreleri atama yapılacak mevcut işlemcilerin hafıza miktarı, her bir görevin işlemciler üzerinde çalışma maliyeti ve görevler arası iletişim maliyeti olmak üzere sistem tasarımından çıkarılmaktadır. Önerilen senaryoda görevler mikroservislere, işlemciler ise kaynaklara (bulut sunucularına) karşılık gelmektedir. Gerekli parametreler çıkarıldıktan sonra yaklaşımı destekleyen araç, optimizasyon algoritmalarını kullanarak verimli dağıtım modelleri üretmektedir. Daha sonra, oluşturulan dağıtım modellerinin uygulanabilirliği bir sonraki adımda değerlendirilmektedir. Oluşturulan dağıtım modelleri tatmin edici değilse, sistem tasarımını analiz etmek ve ilgili araç tarafından sağlanan geri bildirime göre düzeltmek için bir yineleme adımı gerekecektir. Bir dağıtım alternatifinin tatmin edici olması, dağıtım modeli için toplam maliyetin minimize edilmesi (örneğin iletişim ve çalışma maliyetleri) ve bellek

kapasitelerinin verimli kullanımı için beklenen iyileşme oranını karşılaması ile belirlenmektedir. Verimli dağıtım modellerini bulmak, işlem adımlarının birkaç kere yinelemesini gerektirebilir. İlk dağıtım modeli geliştirme ve entegrasyon/test faaliyetlerinde gerçekleştirilip doğrulanmakta ve sonuçlar tatmin edici bir alternatif elde edilinceye kadar tasarımcıya geri bildirilmektedir. Verimli dağıtım alternatifi, her bir dağıtım düğümünün bellek kapasitelerini ihlal etmeden dağıtım modelinin toplam maliyeti (örneğin, iletişim ve yürütme maliyetleri) en aza indirgenerek belirlenmektedir. Verimli dağıtım modelleri bulmak, işlem adımlarının birçok kez yinelemesini gerektirebilir. İlk dağıtım modeli geliştirme ve entegrasyon/test faaliyetleri sırasında gerçekleştirilmekte, sonuçlar tasarımcıya geri bildirilmekte ve tasarım tatmin edici bir alternatif elde edilene kadar revize edilmektedir.

Aşağıdaki bölümlerde, önerilen yaklaşımın uygulanması için tanımlanan somut aktiviteler açıklanmaktadır. Her bir alt bölümde, yaklaşım için gerekli olan metamodellerin tasarımı ele alınmaktadır. Ayrıca birbirine bağlı metamodeller arasındaki ilişkiler açıklanmaktadır.

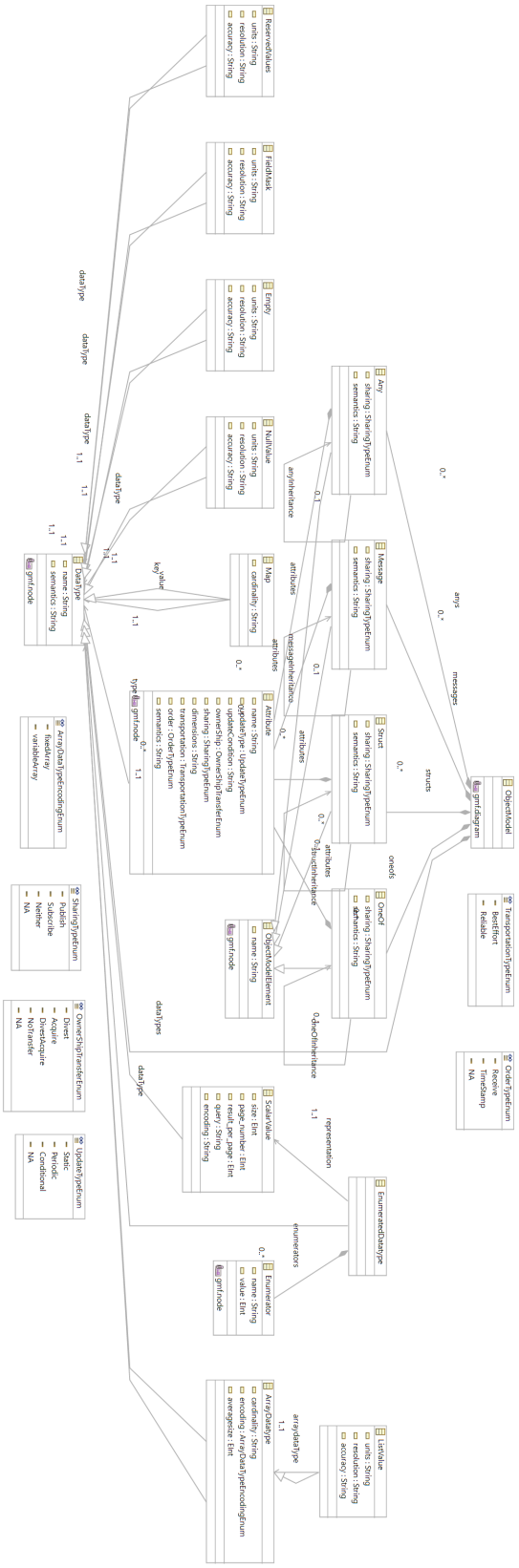
6. GELİŞTİRİLEN METAMODELLER

6.1. Mikroservis Veri Değişim Modeli (Microservice Data Exchange Model)

Mikroservis Veri Değişim Modeli, mikroservisler arasında veri değişimini sağlamak için gerekli olan veri modelini tanımlamaktadır. Mikroservislerin veri değişimi yapması için birbirleri ile haberleşmesi gerekmektedir. Mikroservislerin haberleşmesi için kullanılan protokoller incelendiğinde gRPC [85], REST [89], graphQL [90], publish/subscribe [91] gibi bir çok iletişim altyapısının yer aldığı görülmektedir. Önerilen yaklaşımda, tüm haberleşme yöntemlerinin uygulanabilirliğini sağlamak amacıyla en geniş kapsamlı iletişim altyapısının veri modeli araştırılmıştır. Bu bağlamda veri türü açısından en geniş kümeyi tanımlayan altyapı gRPC olarak belirlenmiştir. gRPC veri modelinde bir çok veri tipi olduğu için bu modelin ecore diyagramı da oldukça karmaşıktır. Graphical Modeling Framework kullanılarak tasarlanan bu modelin ecore diyagramı Şekil 6.1.'de verilmektedir.

Google tarafından geliştirilen yüksek performanslı uzak prosedür çağrısı (high performance Remote Procedure Call) olarak adlandırılan gRPC protokolü herhangi bir ortam üzerinde çalışabilen modern açık kaynak RPC çerçevesidir. Yük dengeleme, izleme, sağlık kontrolü ve kimlik doğrulama için tak-ve-çalıştır (plug-and-play) desteği sayesinde veri merkezleri içindeki ve karşısındaki servisleri etkin bir şekilde bağlayabilmektedir. gRPC, protokol arabelleklerini (protocol buffers) hem Arayüz Tanım Dili (IDL) hem de temel mesaj değişim formatı olarak kullanabilmektedir. gRPC'de kullanılan son sürüm olan proto3'de mesaj formatında arama isteğini tanımlamak için Message Type, .proto dosyasında yer alan değişken türlerinin kapsadığı sınıf ise *Scalar Value Type* olarak adlandırılmaktadır. Bu yüzden veri değişim modelinde yer alan *float*, *int32*, *uint32*, *sint64* gibi değişken türleri *Scalar Value* sınıfından genişletilmektedir. Ayrıca diğer veri değişim modellerinde olduğu gibi enum türünden veri türleri de bu protokolde tanımlanabilmektedir. *ReservedValue*, *NullValue*, *Map* ve *ArrayDataType* gibi farklı veri tipleri de DataType sınıfından genişletilmektedir. *Message Type* ile birlikte herhangi bir JSON nesnesini temsil eden Struct, .proto tanımları olmadan

mesajları gömülü tip olarak kullanmaya izin veren Any ve çok fazla alana sahip bir mesajda aynı anda en fazla bir alanın ayarlanacağı bir yerde kullanılan Oneof özelliği ObjectModelElement sınıfından türetilmektedir.



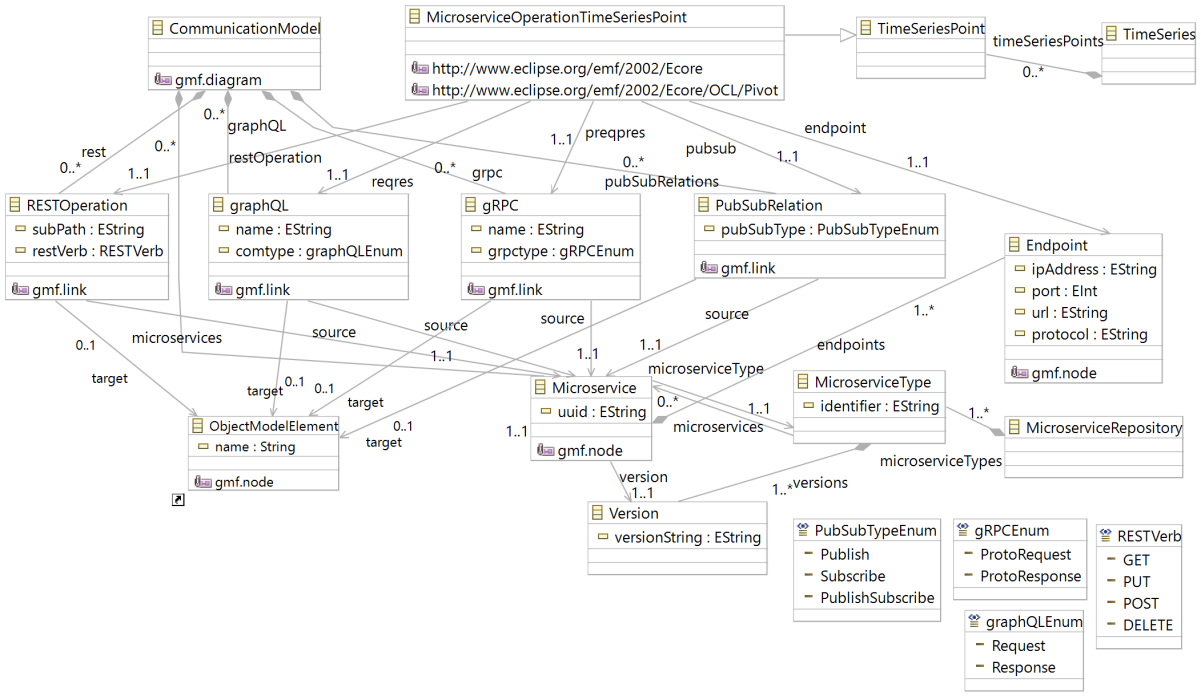
Şekil 6.1. Mikroservis Veri Değişim Modeli

6.2. Mikroservis Tanımlama Modeli (Microservice Definition Model)

Mikroservis Tanımlama Modeli uygulama katılımcıları olarak tanımlanan mikroservislerden oluşmaktadır. Mikroservisler tüm sistemin çalışması için gerekli olan temel yapıtaşıdır. Şekil 5.2. ve Şekil 5.3.'de verilen senaryoda PassengerManagement, DriverManagement, Billing, Payment, Notification gibi servisler mikroservislerdir. Mikroservisler çeşitli iletişim protokollerini kullanarak birbirleriyle iletişim kurduğundan, bu modelin bileşenleri (*Microservice*, *MicroserviceType*, *Version*, *MicroserviceRepository*) Mikroservis İletişim Modeli üzerinde tanımlanmaktadır.

6.3. Mikroservis İletişim Modeli (Microservice Communication Model)

Mikroservis İletişim Modeli, veri değişimine dayalı mikroservislerin birbirleri ile iletişim kurmaları halinde bağlantı kanalının kurulduğu modeldir. Mikroservis Tanımlama Modeli'nde tanımlı olan mikroservisler, birbirleri ile Mikroservis Veri Değişim Modeli'nde tanımlanan veri nesnelerini kullanarak haberleşmektedirler. Çeşitli haberleşme yöntemlerine sahip mikroservis iletişim modelinin ecore diyagramı Şekil 6.2.'de gösterilmektedir.



Şekil 6.2. Mikroservis İletişim Modeli

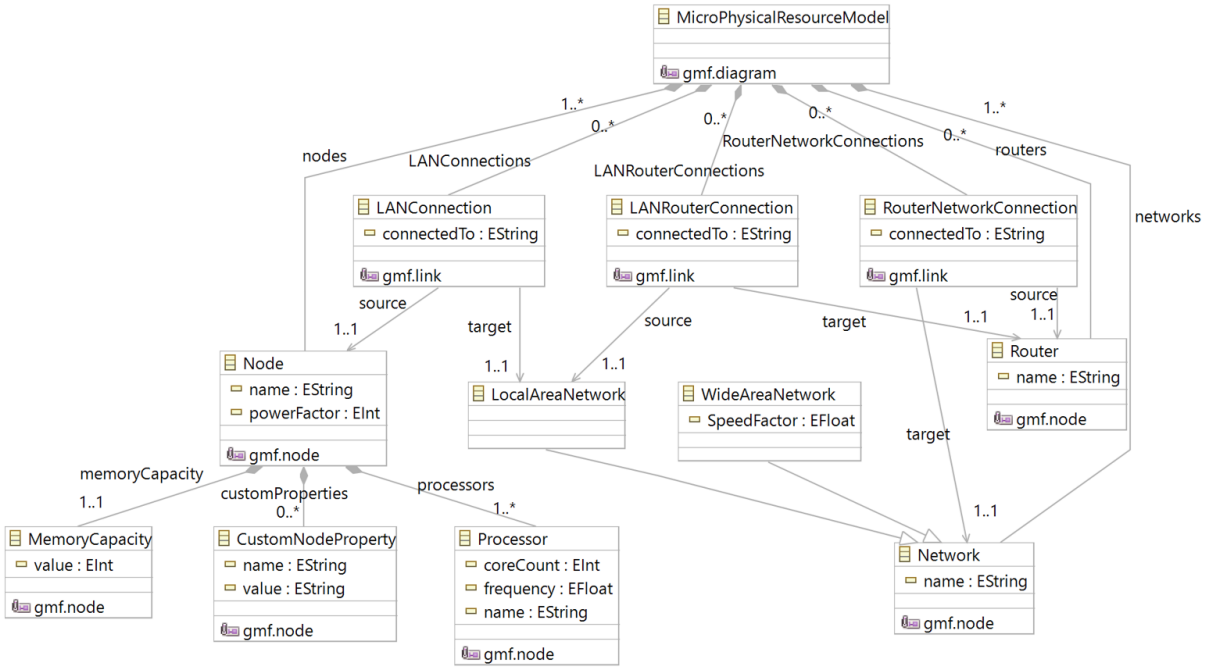
Şekilde görüldüğü gibi mikroservisler için tanımlanan iletişim yöntemleri RestOperation [89], graphQL [90], gRPC [85] ve yayınlama/abone olma (pub/sub relation) [91] şeklinde ayrılmaktadır. Bunlar arasından mikroservis konseptine en uygun olan iletişim türü gRPC'dir. Farklı teknolojilerde ve farklı programlama dillerinde geliştirilmiş birçok mikroservis inşa edildiğinde servis arayüzlerini ve altta yatan mesaj değişim formatını standart bir şekilde tanımlamak oldukça önemlidir. gRPC'de protokol arabellekleri (protocol buffers) kullandığından bu protokol servis iletişimlerini standartlaştırmak için güçlü bir iletişim yöntemidir. Bu yüzden de mikroservisler arasındaki iletişimi inşa etme için mevcut durumda en geçerli çözüm olarak bilinmektedir.

gRPC'den sonra en fazla kullanılan iletişim yöntemlerinden birisi olan yayınla/abone ol (publish/subscribe) ilişkisi, mikroservis mimarilerinde ve sunucusuz mimarilerde servisler arası iletişimin eş zamansız (asenkron) halini ifade etmektedir. Bir pub/sub modelde konuya yayınlanan herhangi bir mesaj, o konuya abone olan tüm servislere iletilmektedir. Olay tabanlı mimariye (event-based architecture) sahip mikroservislere uygun olan pub/sub modeli performans, güvenilirlik ve ölçeklenebilirliği artırmak amacıyla uygulamaları ayırmak için

de kullanılabilmektedir [91]. Durum çalışması üzerinden pub/sub ilişkisine örnek verilecek olursa *PassengerManagement* mikroservisi *Passenger* nesnesini yayınlayabilir, *Driver* nesnesine ise abone olabilir. Benzer şekilde *DriverWebUI* mikroservisi *Driver* nesnesini yayınlarken, *Passenger* nesnesine abone olmaktadır.

6.4. Mikroservis Altyapı Modeli (Microservice Infrastructure Model)

Mikroservis Altyapı Modeli'ni tasarlama aktivitesinde var olan fiziksel kaynakların işlem gücü, bellek kapasiteleri ve düğümler arasındaki ağ bağlantıları tanımlanmaktadır. Yazılımsal iş parçalarının tanımlanması ile fiziksel kaynakların tasarımı birbirlerinden tamamen bağımsız olduğu için bu model diğer metamodellerle paralel bir şekilde tasarlanmıştır. Yani herhangi bir modele ihtiyaç duymadan farklı uygulamalarda da çalışabilmekte ve farklı ekipler tarafından geliştirilebilmektedir. Altyapı modelini kurmak için tasarlanan fiziksel kaynaklara örnek olarak mikroservislerin yerleştirilip çalıştırılacağı belirli parametrelere sahip bir veya daha fazla düğüm tanımlanabilmektedir. Tanımlanan bu düğümlerin bellek kapasiteleri 1024 MB, 2048 MB, vb. ve 2.3 MHz frekansta çalışan 4 çekirdekli 2 işlemciye vb. özelliklere sahip olacak şekilde ihtiyaca göre güncellenebilmektedir. Düğümlerin tümü ortak bir yerel ağa bağlanarak düğümler arasında iletişim için homojen bir bağlantı ağı kurulabilmektedir. Bununla birlikte farklı frekansa, çekirdek sayısına ve bellek kapasitelerine sahip düğümler tasarlanarak farklı iletişim performanslarına sahip yerel alan ve geniş alan ağları tanımlanıp düğümleri ağ üzerinde farklı noktalara dağıtarak heterojen bir bağlantı modeli kurmak mümkündür. Özellikleri bahsedilen bu metamodelin tasarımı Şekil 6.3.'te gösterilmektedir.



Şekil 6.3. Mikroservis Altyapı Modeli

MicroPhysicalResourceModel, bir fiziksel kaynak modelini temsil eden ana sınıftır. Bir fiziksel kaynak modelinde Node sınıfının olgularıyla temsil edilen bir veya birden fazla düğüm bulunabilmektedir. Node sınıfında name ve powerFactor olmak üzere iki özellik bulunmaktadır. Bu özelliklerden name, düğümün kimlik bilgisini içerirken powerFactor düğümün diğer düğümler ile göreceli olarak işlem gücünü tanımlamaktadır. Bir düğümün bir veya daha fazla işlemci birimine sahip olması, *Processor* sınıfı ile gerçekleşmektedir. Bu sınıfta *name*, *frequency* ve *coreCount* özellikleri ile bir işlemci birimi oluşturulabilmektedir. *Name* işlemci biriminin sembolik adını temsil ederken, *coreCount* işlemci biriminin sahip olduğu çekirdek sayısını, *frequency* ise işlemcinin MHz cinsinden frekansını temsil etmektedir. *Processor* sınıfına benzer şekilde bir düğümün farklı bellek kapasiteleri olabilir. *MemoryCapacity* sınıfında yer alan value niteliği ile düğümün sahip olduğu bellek miktarı MegaByte cinsinden tanımlanmaktadır. Mikroservis Altyapı Modeli'nde her bir düğüm için tanımlanan işlemci gücü, bellek kapasitesi gibi özelliklerin yanısıra kullanıcıların isteğine bağlı olarak tanımlayabileceği (disk kapasitesi gibi) nitelikler için *CustomNodePropoperty* sınıfı oluşturulmaktadır. Bu sınıfta yer alan name ve value nitelikleri ile gösterilen isim-değer ikilileri

tanımlanabilir. Örneğin bir düğümün disk kapasitesi "diskCapacity" adı ve “240GB” değeri ikilisi ile tanımlanabilmektedir. Metamodelde bulunan *Network* sınıfı yerel alan ağlarını temsil eden *LocalAreaNetwork* ve *WideAreaNetwork* sınıflarının soyut ata sınıfıdır. Bir fiziksel altyapıda bir veya daha çok ağ tanımı yapılabilmektedir. *LocalAreaNetwork WideAreaNetwork*'e göre daha hızlı olduğundan dolayı *WideAreaNetwork* sınıfına eklenen *speedFactor* özelliği ile ağın LAN'a göre ne kadar yavaş olduğu belirlenmektedir.

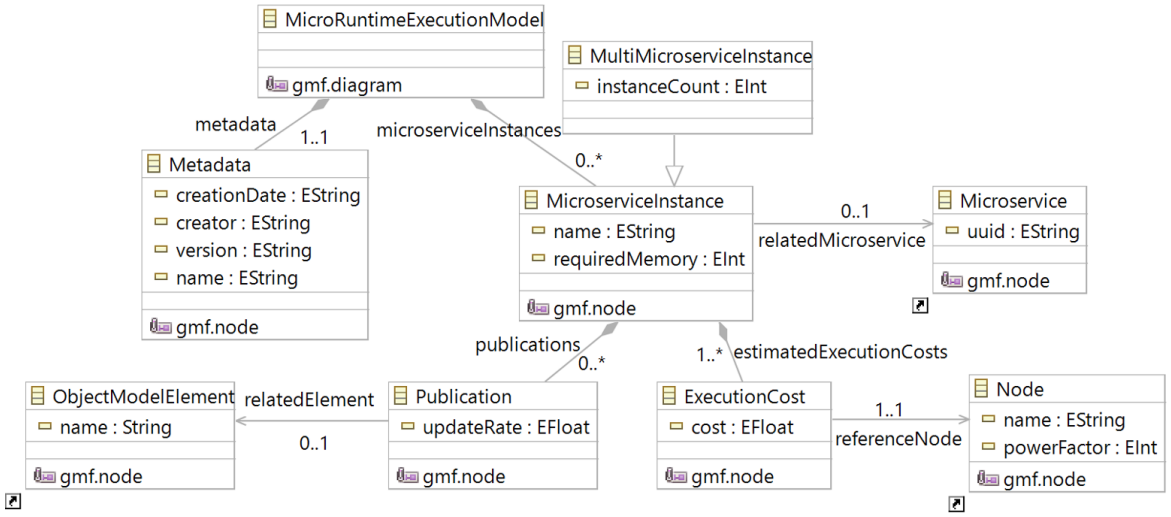
LANConnection sınıfı bir düğümün bir yerel alan ağına bağlantısını tanımlarken Router sınıfı, ağları birbirlerine bağlamak için kullanılan yönlendiricileri tanımlamaktadır. *LANRouterConnection* sınıfı bir yerel alan ağının yönlendiriciye bağlantısını temsil ederken, *RouterNetworkConnection* sınıfı bir yönlendiricinin bir ağa bağlantısını temsil etmektedir.

6.5. Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli (Microservice Runtime Execution Configuration Model)

Oluşturulacak sistemin yapısal özellikleri, Mikroservis Veri Değişim Modeli, Mikroservis Tanımlama Modeli, Mikroservis İletişim Modeli ve Mikroservis Altyapı Modeli'nin kurulması ile tamamlanmaktadır. Bu yapısal tanımlamaları kullanarak sistemde hangi bileşenden kaç adet bulunduğu, bileşenlerin veri modeli elemanlarının ne kadar sıklıkla güncellendiği, her bir elemanın her bir hedef düğüm üzerindeki işletim maliyeti gibi koşum zamanına dair bilgilere de ihtiyaç duyulmaktadır. Bu bağlamda geliştirilen yaklaşımın önemli bir parçası olan ve çalışması için önceki bölümlerde bahsedilen tüm modellere ihtiyaç duyulan çalışma zamanı yürütme konfigürasyon modeli tasarlama aktivitesi tanımlanmaktadır. Örneğin kullanıcı, *DriverManagement* isimli mikroservis saniyede beş kez *Vehicle* nesnesini güncelleyecek şekilde bir tasarım yapabilmektedir. İşlem gücüne bağlı olarak her bir *DriverManagement* olgusu için çalışma maliyeti ölçeklendiriliş olarak bir düğüm için 10 üzerinden 5, başka bir düğüm için 10 üzerinden 8 şeklinde tanımlanabilmektedir.

Çalışma zamanı yürütme konfigürasyonlarını modellemek için gerekli elemanları içeren metamodel Şekil 6.4.'de verilmektedir. Burada *MicroRuntimeExecutionModel* sınıfı metamodelin kök sınıfıdır ve bir çalışma zamanı yürütme konfigurasyonunu tanımlamaktadır. Bir

çalışma zamanı yürütme konfigürasyonu, bir *Metadata* ve bir grup *MicroserviceInstance* örneğinden oluşmaktadır.



Şekil 6.4. Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli

Çalışma zamanı yürütme konfigürasyonunu tanımlayıcı nitelikleri içeren sınıf *Metadata* sınıfıdır. Bu nitelikler *name*, *version*, *creator* ve *creationDate* olarak belirlenmektedir. *MicroserviceInstance*, Mikroservis Tanımlama Modeli'nde tanımlanan mikroservislerin bir örneğini göstermektedir. Örneğin *DriverManagament* mikroservisini başlatan bir mikroservis örneğinin adı "BMW X6" olabilir.

MicroserviceInstance sınıfına ait *requiredMemory* özelliği mikroservis örneğinin çalışma esnasında ihtiyaç duyacağı tahmini hafıza miktarını temsil etmektedir. Çalışma maliyetine benzer şekilde *requiredMemory* özelliği de tasarım zamanında tahmin edilebilmektedir. Çalışma konfigürasyonu esnasında örnek sayısını tanımlayan nitelik ise *instanceCount* niteliğidir. Bu özelliğin eklenme sebebi, bir çalışma konfigürasyonunda aynı mikroservis örneğinden birden fazla olabilme ihtimalinin bulunmasıdır. Örneğin bir taksi çağırma senaryosunda aynı anda birden fazla aracın yolcu alması kaçınılmazdır, böylece *Driver* ve *Passenger* mikroservis örneklerinden birden fazla gerekmektedir. Saf modelleme yaklaşımı, tasarımcıyı her örnek için yürütme modeline bir mikroservis örneği eklemeye zorlamaktır, ancak bu kesinlikle kullanıcı dostu bir yaklaşım değildir. Bu yüzden çalışma zamanı yürütme konfigürasyonundaki örnek

sayısını içeren bir *instanceCount* özniteliğine sahip *MultiMicroserviceInstance* sınıfı adında bir sınıf tanımlanmaktadır. Bu sınıfı kullanarak tasarımcı, tek tek 100 *MicroserviceInstance* sınıfı eklemek yerine, *instanceCount* 100 olarak ayarlanmış senaryoya bir *MultiMicroserviceInstance* sınıfı ekleyebilmektedir.

MicroserviceInstance sınıfına ait *RelatedMicroservice* ilişkisi Mikroservis Tanımlama Modeli'nde tanımlanan bir *Microservice* ile *MicroserviceInstance* arasındaki ilişkiyi temsil etmektedir. *MicroserviceInstance*, güncelleme sıklığını temsil eden sıfır ya da daha fazla iletişim sınıfına sahiptir. İletişim sınıflarında yer alan *updateRate* özelliği ise bir saniyede bir mikroservis örneğinin kaç kere güncellendiğini temsil etmektedir.

6.5.1. Mikroservis Çalışma Zamanı Parametrelerini Tahmin Etme

Bu tez çalışması kapsamında önerilen yaklaşım, uygulanabilir dağıtım tasarımı alternatiflerini erkenden belirlemek için yazılım yaşam döngüsünün başlarında kullanılabilmektedir. Bununla birlikte, yaklaşım, sistemin dağıtım mimarisini geliştirmek için yaşam döngüsünün ilerleyen dönemlerinde de benimsenebilmektedir.

Bu aynı zamanda, servislerin hali hazırda geliştirildiği ve bellek ve CPU gereksinimlerinin servisleri fiilen çalıştırarak yüksek doğrulukla ölçülebildiği entegrasyon ve test gibi sonraki aşamaları da içermektedir. Önerilen yaklaşımın kullanıldığı aşamalardan bağımsız olarak, beklenen iş yükü ve servis özelliklerine dayalı olarak, servisin tahmini CPU ve bellek gereksinimlerini kullanmak bu parametrelerin deneysel ortamda net olarak belirlenmesine yardımcı olmaktadır. Servis gerçek ortamda geliştirilmeden önce mikroservislerin kaynak gereksinimlerini tahmin etmek için aşağıdakileri içeren kapsamlı bir analiz gerekmektedir:

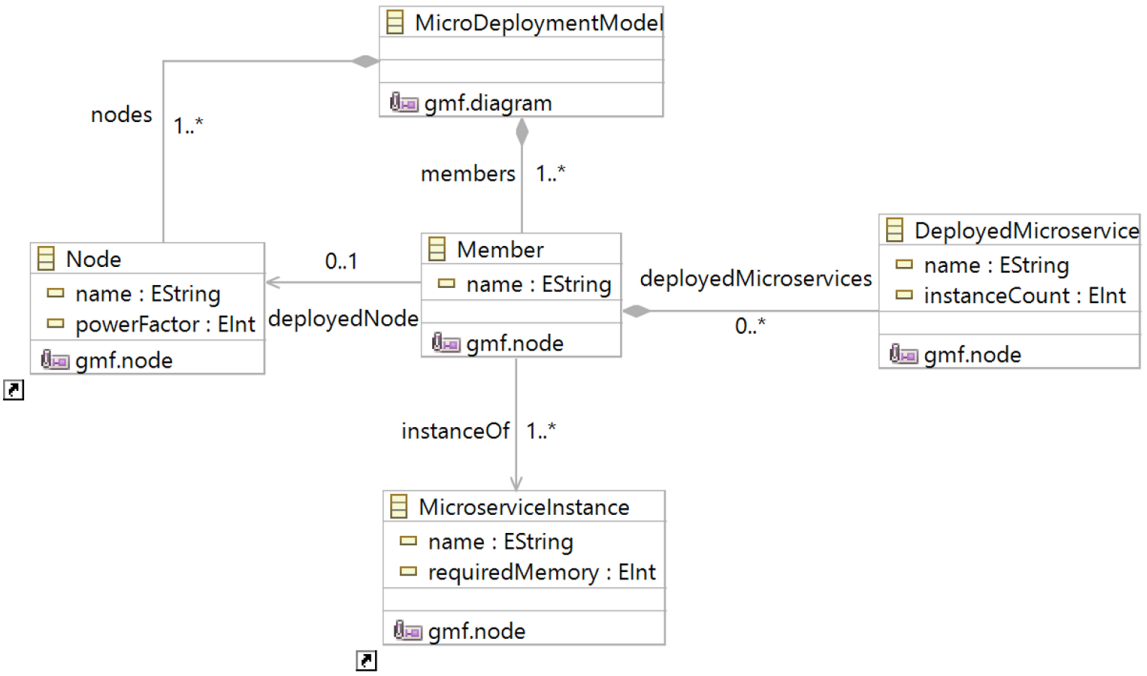
- İş analizine dayalı ayrıntılı kullanım durumu ve kullanım analizi
- Servis kalite gereksinimlerinin analiz edilerek belirlenmesi
- Bilgi işlem, alan ve ağ oluşturmanın belirli maliyetleri ve özellikleri

- Benzer durum çalışmaları için önceki dağıtım modellerinden elde edilen geçmiş deneyimler

Servislerin kaynak gereksinimleri, geliştirme yaşam döngüsü boyunca sürekli olarak ölçülmeli ve izlenmelidir. Servislerin dağıtımı esnasındaki kaynak kullanımının manuel olarak sürekli olarak ölçülmesi ve izlenmesi, çaba ve zamanlama açısından izlenebilir olmadığından, ölçüm ve izleme süreci otomatikleştirilmeli ve otomatikleştirilmiş test çerçeveleri kullanılarak mümkün olan en kısa sürede CI/CD (Continous Integration/Continous Delivery) sürecine entegre edilmelidir.

6.6. Mikroservis Dağıtım Modeli (Microservice Deployment Model)

Mikroservislerin dağıtımı için gerekli tüm fiziksel ve yapısal özellikler tasarlandıktan sonra sisteme dahil olan her bir mikroservis üyesinin düğümlerden birine yerleştirilmesi için bir dağıtım aktivitesi tasarlanmaktadır. *MicroDeployment* adı verilen metamodelde altyapı modelinde tanımlanan düğüm (*Node*) ve üye (*Member*) arasında 1 veya daha çok ilişki vardır. Bu da bir ya da daha fazla üyenin düğümlerden birine aktarıldığını temsil etmektedir. Ayrıca Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'nde tanımlanan bir ya da daha fazla mikroservis örneği de bir *Member* üzerinde dağıtılabilmektedir. Açıklaması verilen *MicroDeployment* metamodeli Şekil 6.5.'de gösterilmektedir.



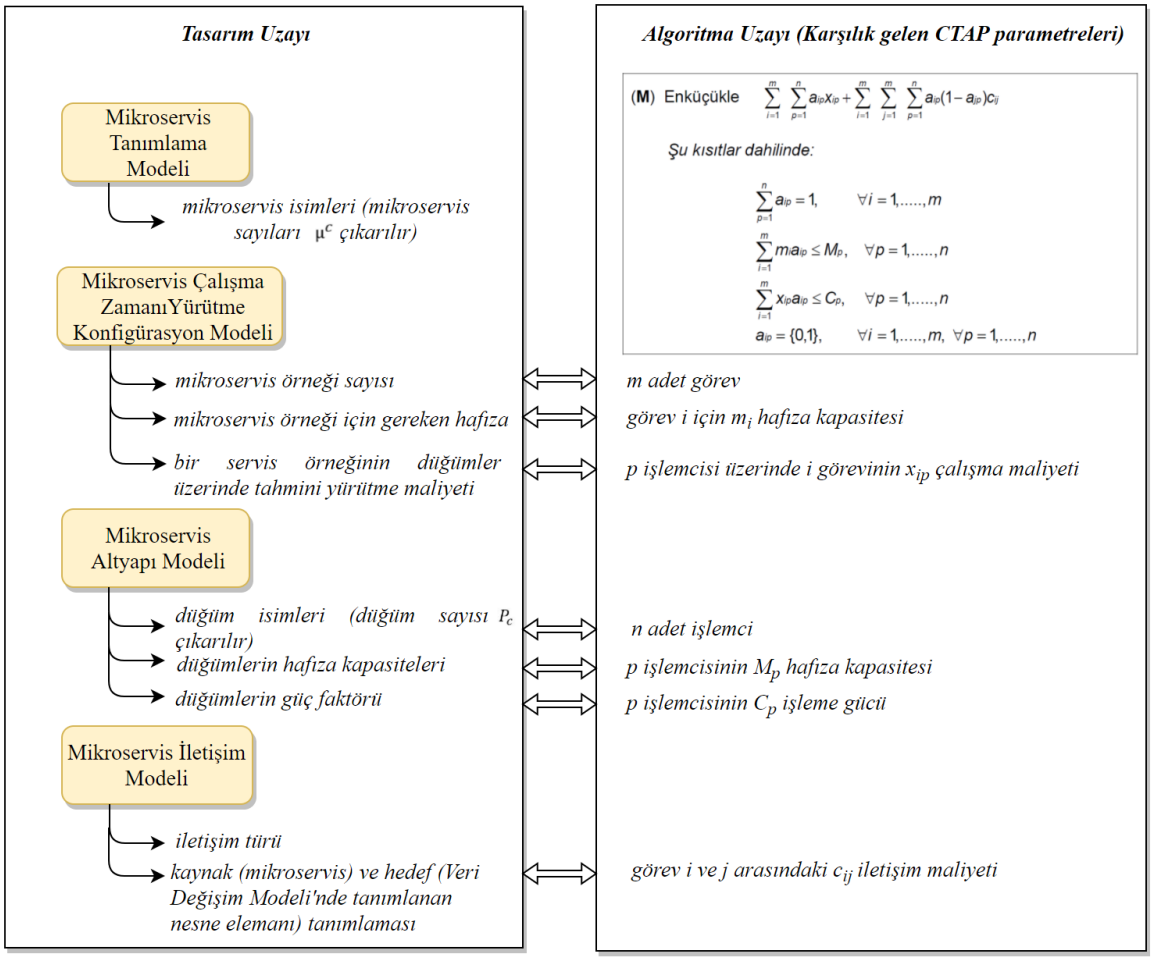
Şekil 6.5. Mikroservis Dağıtım Modeli

7. KAYNAK TAHSİS ETME YÖNTEMİ İÇİN GİRDİ PARAMETRELERİ ÜRETME

Uygulama ortamının tasarımı, önceki bölümlerde açıklandığı gibi yapısal özellikler, fiziksel altyapı ortamının hazırlanması ve uygulama elemanlarının yürütme konfigürasyonları tanımlanarak tamamlanmaktadır. Bu bölümde sisteme girdi olarak verilen hafıza gereksinimleri, mikroservis örneklerinin her bir düğüm üzerindeki çalışma maliyetleri, mikroservisler arasındaki iletişim maliyetleri ve hesaplama güçleri gibi tasarımda tanımlı kısıtlar göz önüne alınarak mikroservis örneklerinin düğümlere tahsis edilmesi konusu ele alınmaktadır. Çalışma kapsamında ele alınan tahsis etme algoritmasının Kapasite Kısıtlı Görev Atama Problemi-CTAP (Capacitated Task Assignment Problem) ile doğrudan eşleştiği görülmektedir. Aşağıdaki alt bölümde CTAP algoritmasının önerilen yaklaşıma uyarlanma adımları anlatılmaktadır.

7.1. Model Tasarımından Elde Edilen Parametreler ile CTAP Parametrelerinin Eşleştirilmesi

Mikroservislerin dağıtımı için gerekli olan tüm modeller oluşturulduktan sonra bu modellerden parametre çıkarımının yapılması gerekmektedir. Bunun sebebi, mikroservislerin en uygun düğüme yerleştirme işlemi yapılırken servislerin depolanması için gerekli hafıza, servislerin düğümler üzerindeki çalışma maliyeti, düğümlerin hafıza kapasiteleri gibi bir çok parametreye ihtiyaç duyulmasıdır. Mikroservislerin dağıtımı için ele alınan parametreler literatürde yer alan Kapasite Kısıtlı Görev Atama Problemi (Capacitated Task Assignment Problem) ile eşleşmektedir [19]. Bölüm 2.4.'de bahsedilen CTAP parametreleri ile model tasarımlarından çıkarılan parametrelerin eşleştirilmesi Şekil 7.1.'de gösterilmektedir.

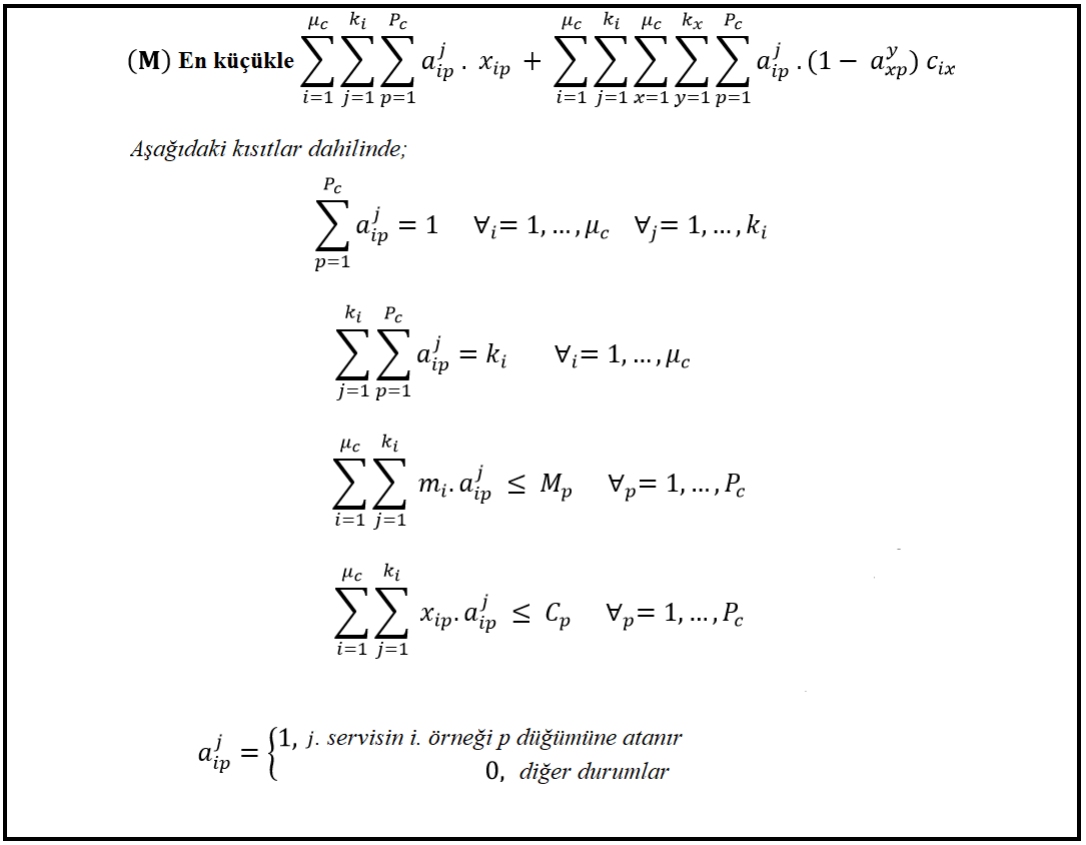


Şekil 7.1. Önerilen yaklaşım için tasarım ve algoritma uzayları arasındaki eşleşme

Önerilen yaklaşım için karşılık gelen CTAP parametreleri aşağıdaki şekilde tanımlanmaktadır: Mikroservis Tanımlama Modeli'nde oluşturulan mikroservisler CTAP'de görev (task) sınıfına karşılık gelmektedir. Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'nde tanımlı her bir mikroservis örneği tek bir göreve karşılık gelmektedir. Mikroservislerin bir özelliği olan mikroservis örneği sayısı (*instanceCount*), bir mikroservis (m) için görev sayısını temsil etmektedir. Mikroservis Altyapı Modeli'nde tanımlı her düğüm (node) bir işlemciye (processor), (*memoryCapacity*) özelliği ile ifade edilen her bir p düğümünün hafıza kapasitesi, CTAP'de p işlemcisinin M_p hafıza kapasitesine karşılık gelmektedir. p düğümünün güç faktörü (*powerFactor*) özelliği p işlemcisinin hesaplama gücüne karşılık gelmektedir. Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'nde tanımlı *ExecutionCost* sınıfının *cost* özelliği her bir p işlemcisi üzerinde çalışan i görevinin çalışma maliyetini (X_{ip})

temsil etmektedir. Bu sınıf her bir p düğümü üzerinde her bir mikroservis örneğinin çalışma maliyetini tanımlamak için Mikroservis Altyapı Modeli'nde tanımlı *Node* sınıfına bağlıdır. Son parametre olan c_{ij} parametresi ise i ve j görevleri için iletişim maliyetini tanımlamaktadır. Önerilen yaklaşımda iletişim maliyeti, Mikroservis Veri Değişim Modeli'nde tanımlı veri nesnelerinin boyutu ile Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'nde tanımlı veri güncellenme sıklığının (*updateRate*) çarpımı ile hesaplanmaktadır. Bu tez çalışmasının sonunda yer alan Ek A'da iletişim ve çalışma maliyetlerinin hesaplanması için tasarlanan sözde kod verilmektedir.

Yaklaşımın problem tanımlaması Şekil 7.2.'de verilmektedir. CTAP tanımlamasına benzer şekilde sistemde μ^c adet mikroservis bulunmakta ve μ_i, $i.$ mikroservisi temsil etmektedir. m_i birim hafızaya sahip $i.$ mikroservis toplamda k_i örneğe sahiptir. μ_i^j ise i mikroservisinin $j.$ örneğini ifade etmektedir. Ortamda aynı zamanda P_c adet benzersiz düğüm (işlemci) yer almaktadır ve p düğümü M_p birim hafıza ve C_p birim işleme kapasitesine sahiptir. p düğümü üzerinde i mikroservisinin çalışma maliyeti x_{ip}'dir. i ve j mikroservislerinin arasındaki toplam iletişim maliyeti eğer bu mikroservisler farklı düğümlere atanmışlarsa c_{ij} olarak ifade edilir. Servislerin aynı düğüme atanması ile iletişim maliyetleri 0 kabul edilir. Bu optimizasyon probleminin amacı mikroservisleri düğümler üzerinde iletişim ve çalışma maliyetlerinde hafıza ve hesaplama gücü gibi kısıtları aşmadan minimuma indirgemektedir. Problemin karar değişkeni a_{ip}^j eğer μ_i^j bir düğüme atanmışsa 1, atanmamışsa 0 olarak işlenmektedir.



Şekil 7.2. Önerilen yaklaşımın matematiksel modellemesi

Şekil 7.2.'de verilen M matematiksel modelinin amacı şu şekilde açıklanabilir: "Mikroservisleri, düğümlere çalıştırma maliyetleri ve iletişim maliyetlerinin toplamı asgari olacak şekilde ata. Eğer iki servis aynı düğüme atanmışsa, aralarındaki iletişim maliyetinin sıfır olduğunu varsay. Atamaları yaparken her düğüme atanan servislerin toplam bellek ihtiyacının düğümün bellek miktarını aşmadığına ve toplam işlem gücü ihtiyacının da düğümün işlem gücünü aşmadığına emin ol." Bu yaklaşımla, yukarıdaki teorik modele benzer bir şekilde, mikroservisler modellenerek sanal makineler ve kapsayıcılar üzerine en iyiye yakın şekilde yerleştirme işlemleri yapılmaktadır. Bu problem literatürde NP-zor olarak bilinen ve en iyi çözümü doğrusal zamanda bulunamayan bir problemdir. Bu nedenle verimli dağıtım alternatifleri üreten farklı optimizasyon algoritmaları uygulanmakta, fakat bu algoritmaların en iyi çözümü sunacağı garantisi verilmemektedir. Bu algoritmalar yardımı ile birbirleriyle sık haberleştiği bilinen mikroservislerin mümkün mertebe aynı düğümler üzerine yerleştirilmesi

Çizelge 7.1. Tasarımdan türetilen CTAP uyarlamalı parametreler

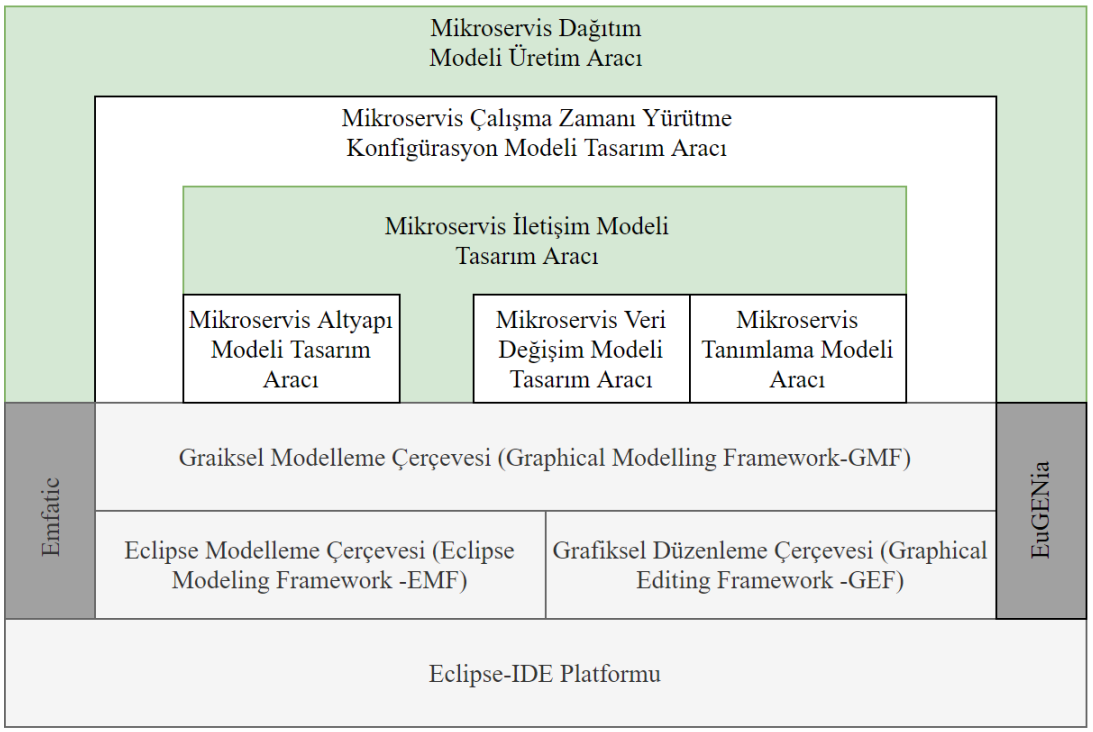
CTAP Parametresi	Tasarımdan elde edilen parametreler
μ_c	Mikroservis Tanımlama Modeli'nde tanımlı *mikroservis* sayısı
k_i	Mikroservis Tanımlama Modeli'nde tanımlı her bir *microservice i* için Çalışma zamanı yapılandırma modelinde tanımlı *mikroservis örneği* sayısı
P_c	Altyapı Modeli'nde tanımlı *düğüm (işlemci)* sayısı
a_{ip}^{j}	$i.$ mikroservisin $j.$ örneğinin *Düğüm p*'ye atanması
M_p	Altyapı Modeli'nde tanımlı *Düğüm p*'nin hafıza kapasitesi
C_p	İşlemcinin işlem gücüne karşılık gelen *Düğüm p*'nin *güç faktörü (powerfactor)*
m_i	Çalışma Zamanı Yürütme Yapılandırma modelinde tanımlı *MicroserviceInstance i*'nin depolanması için gerekli hafıza miktarı
X_{ip}	*MicroserviceInstance i*'nin *Düğüm p* üzerindeki tahminin çalışma maliyeti
c_{ij}	Farklı düğümler üzerindeki *i* ve *j* mikroservisleri arasındaki iletişim maliyeti. Eğer iki mikroservis aynı düğüme atanmışsa iletişim maliyeti ihmal edilir. Farklı düğümlerdeki maliyet aşağıdaki özellikler kullanılarak hesaplanır: • Publish/Subscribe iletişim modeli için çalışma zamanı yapılandırma modelinde tanımlı yayınlar (publications) • Mikroservis iletişim modelinde yer alan gRPC, Publish/-Subscribe, graphQL, Rest Operation gibi bir çok iletişim modelinin tanımlandığı ilişkiler • Veri değişim modelinde tanımlı nesne modeli elemanları

amaçlanmaktadır. Bu çalışma kapsamında yukarıda bahsedilen parametreler ile benzetim ortamında kullanılan parametrelerin eşleşmiş hali Çizelge 7.1.'de verilmektedir.

Önerilen yaklaşımın temel CTAP'ten en belirgin farkı mikroservisler ve mikroservis örneklerinin tanımıdır. Bir mikroservisin birden fazla örneği olabileceği için CTAP'ten ayrı olarak k_i parametresi kullanılarak bir mikroservis türünden kaç adet mikroservis örneğinin tanımlandığı ifade edilmelidir. Buna göre p düğümü üzerinde her bir μ_i^j örneğinin iletişim ve çalışma maliyetleri a_{ip}^{j} parametre değeri kullanılarak hesaplanmaktadır.

8. Micro-IDE ARAÇ AİLESİ

Bu bölümde Bölüm 5.3.'da tanımlanan yaklaşımı destekleyen araç ailesi geliştirilerek tüm modellerin entegre bir şekilde çalışması sağlanmaktadır. Micro-IDE aracı Bölüm 6.'de tanımlanan metamodellere ve bu metamodellere uygulanan CTAP uyarlamalı yöntemlere dayanmaktadır. Modeller, Eclipse platformu üzerinde eklenti kümeleri halinde çalışmaktadır. Geliştirilen bu eklentiler Eclipse Modeling Framework (EMF), Graphical Modeling Framework (GMF) ve Graphical Modeling Project (GMP) gibi Eclipse Framework eklentileri üzerinde inşa edilmektedir. Eclipse Modeling Framework (EMF), veri modelini tasarlamak ve modele baglı olarak kod ve diger çıktıları üretmek için kullanılan bir Eclipse plugin kümesidir. EMF, metamodel ile gerçek model arasında bir ayrıma sahiptir. Meta-model modelin yapısını tanımlamaktadır. Bir model bu meta modelin somut bir örnegidir. EMF geliştiricinin XMI, Java açıklamaları, UML veya bir XML şeması gibi farklı anlamlarda metamodeller oluşturmasına izin vermektedir. Aynı zamanda varsayılan uygulama, XML Metadata Interchange adlı bir veri biçimini kullanarak model verisinin sürekliligini sağlamaktadır. Graphical Editing Framework (GEF), grafiksel uygulamalar ile ilişkili uç kullanıcı bileşenlerini ve görünümlerini sağlayan bir Eclipse projesidir. GMF ise EMF ve GEF arasında üretici bir köprü saglamayı amaçlayan bir Eclipse modelleme projesidir. Eclipse IDE 2018 sürümü ile ecore modeli oluşturmak kontrol paneli (dashboard) aracılığı ile domain model, domain gen model, graphical def model, tooling def model, mapping model ve diagram editor gen model kolaylıkla türetilmektedir. Ancak bu çalışma bağlamında, ecore modellerinin dönüşümü için Emfatic [92] adlı bir metin editörü kullanılmıştır. Ayrıca .gmfgraph, .gmftool ve .gmfmap gibi ihtiyaç duyulan tüm modellerin Ecore modeli kullanılarak oluşturulması için GMF panosu yerine daha pratik olan EuGENia GMF aracı [93] kullanılmaktadır. Micro-IDE aracının katmanlı mimarisi Şekil 8.1.'de gösterilmektedir.

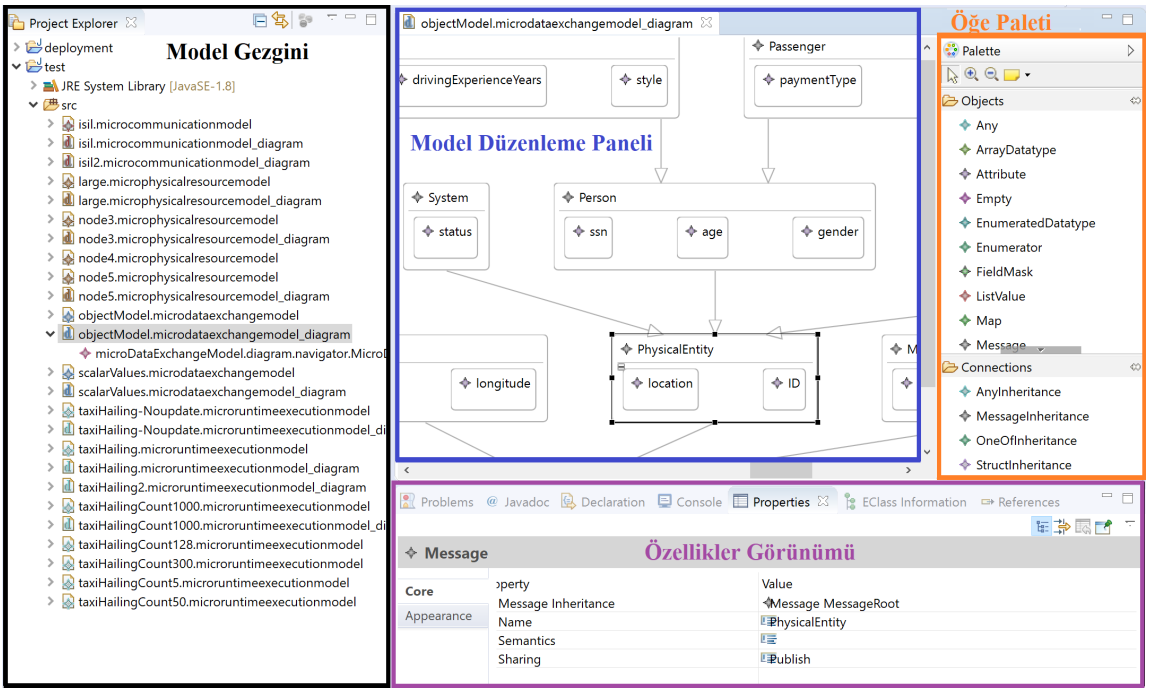


Şekil 8.1. Micro-IDE aracının katmanlı mimarisi

İlerleyen alt başlıklarda seçilen bir durum çalışmasına göre oluşturulan Micro-IDE tasarımından bahsedilerek bu durum çalışması için oluşturulan mikroservis örneklerinin yerleştirildiği dağıtım modeli açıklanmaktadır.

8.1. Araç Mimarisi

Genel bir perspektif olarak Micro-IDE aracı, Şekil 8.2.'de görüldüğü gibi model gezgini (model navigator), model düzenleme bölgesi (model editing pane), ürün paleti (item palette) ve özellikler görünümü (properties view) olmak üzere 5 ana bölümden oluşmaktadır.



Şekil 8.2. Micro-IDE aracının düzenleme bölümleri

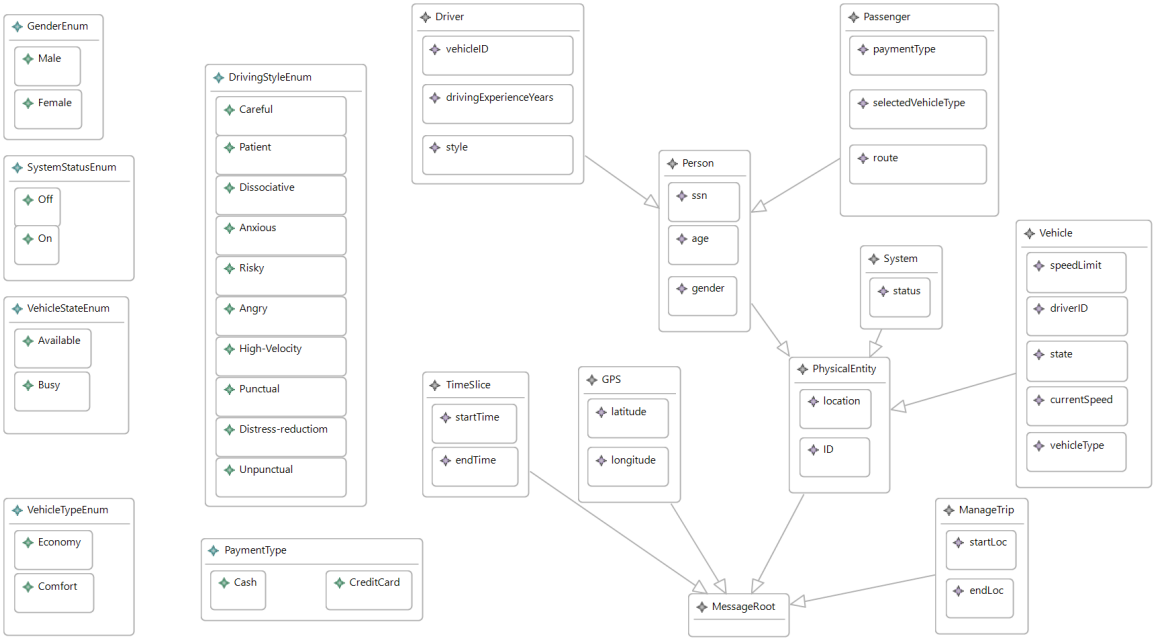
Aracın sol kısmında yer alan model gezgini, oluşturulan metamodelleri ve bu modeller içerisinde kullanılan elemanları göstermektedir. Orta kısımda yer alan model düzenleme bölgesi, mimari tasarımı için gerekli çizim alanını oluşturmaktadır. Altta yer alan özellikler görünümü, model düzenleme bölgesinde çizilen elemanların ve model gezginindeki metamodellerin özelliklerini listelemektedir. Sağda yer alan öğe paleti, model düzenleme bölgesindeki çizimin yapılması için gerekli elemanları sunmaktadır.

8.2. Taksi Çağırma Uygulaması için Micro-IDE Kullanımı

Bu bölümde, son yıllarda mikroservis mimarilerine geçerek birçok ülkede kesintisiz kullanılabilen UBER'in [5] taksi çağırma uygulaması durum çalışması olarak kullanılmaktadır. Bölüm 4.'de detaylı olarak anlatılan bu durum çalışmasının Micro-IDE aracında uygulanması ile oluşturulan modeller ve bu modeller kullanılarak türetilen dağıtım modeli alt başlıklar halinde açıklanmaktadır.

8.2.1. Taksi Çağırma Uygulamasının Veri Değişim Modelinin Tasarlanması-Nesne Modeli ve Veri Tipleri

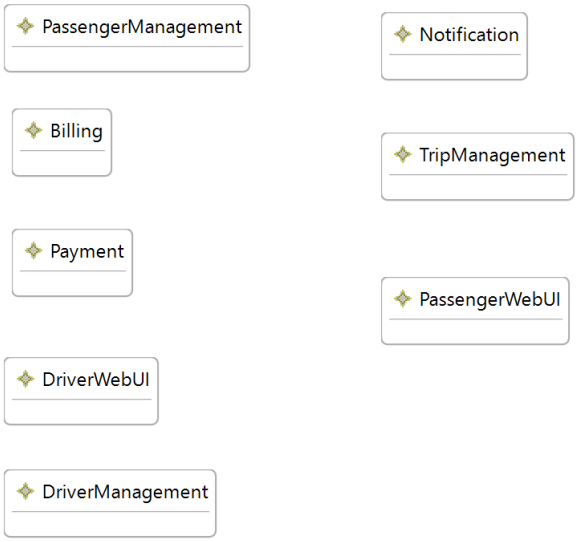
Taksi çağırma uygulamasının veri değişim modeli tasarlanırken, öncelikle bu uygulamada kullanılan nesneler belirlenmektedir. Örneğin Vehicle nesnesi için kullanıcının tercih edebileceği *Economy* ve *Comfort* sınıfları Enum olarak oluşturulmaktadır. Benzer şekilde kullanıcının araç talep etmesi durumunda araçların uygun ya da meşgul olduğunu belirten *VehicleStateEnum* veri tipi oluşturulmaktadır. Aracı kullanan sürücü, hizmeti alan yolcu, ödeme türü, aracından yolcuyu aldıktan sonra gideceği yere kadar bırakma süresi, GPS bilgileri gibi bir çok bilginin yer aldığı model Mikroservis Veri Değişim Modeli'dir. Bu nesneler için gerekli veri tipleri de *Scalar Value* kullanılarak atanmaktadır. Şekil 8.3.'de taksi çağırma uygulaması için tasarlanan veri değişim modeli verilmektedir. Değişken türleri de oluşturulan nesnenin özellikler bölümünden atanmaktadır.



Şekil 8.3. Veri değişim modeli-nesne sınıfları

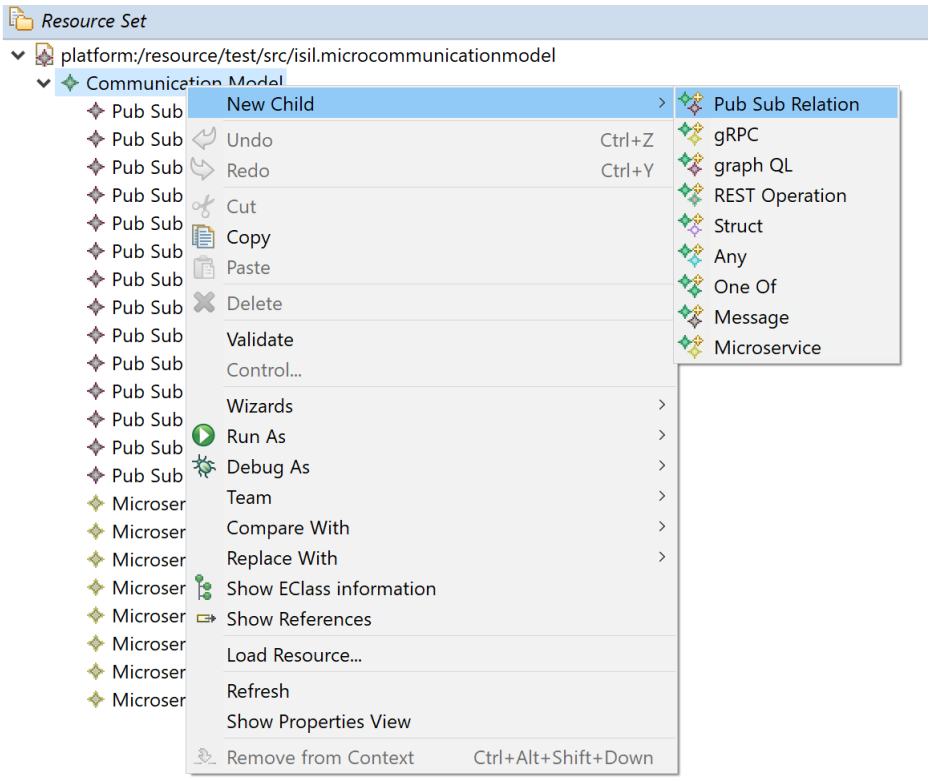
8.2.2. Taksi Çağırma Uygulamasına Ait Mikroservislerin Tanımlanması ve Aralarındaki İletişim Deseninin Kurulması

UBER'in taksi çağırma uygulamasında Şekil 4.1.'de görüldüğü gibi 8 adet mikroservis kullanılmaktadır. Bu mikroservisler Micro-IDE'de tasarlandıktan sonra birbirleri ile iletişimde olan servisler, bu çalışma kapsamında yayınla/abone ol (publish/subscribe) iletişim modeli ile birbirlerine bağlanmaktadır. Servisler arası iletişimi kurmak için Micro-IDE aracı üzerinde mikroservisler Şekil 8.4.'de görüldüğü gibi oluşturulmaktadır.

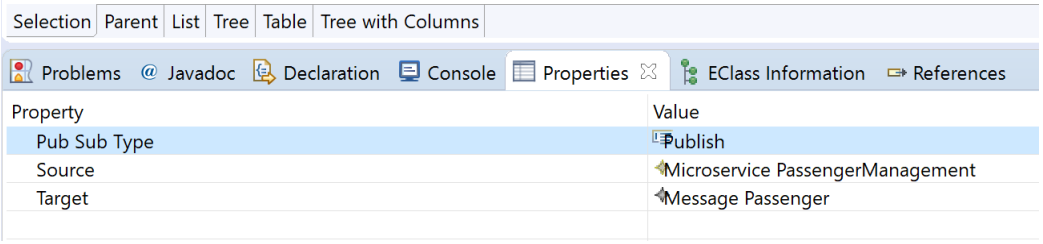


Şekil 8.4. Durum çalışmasına ait mikroservisler

Oluşturulan mikroservisler üzerinde pub/sub ilişkileri Şekil 8.5.'teki gibi kurularak hangi servisin hangi servise üye olduğu ve hangi servisi yayınladığı, oluşturulan pub/sub ilişkinin özellikler kısmından Şekil 8.6.'daki gibi seçilerek her bir mikroservis, iletişimde bulunduğu veri değişim modeli elemanlarına atanmaktadır.



Şekil 8.5. Yayınla/Abone ol ilişkileri



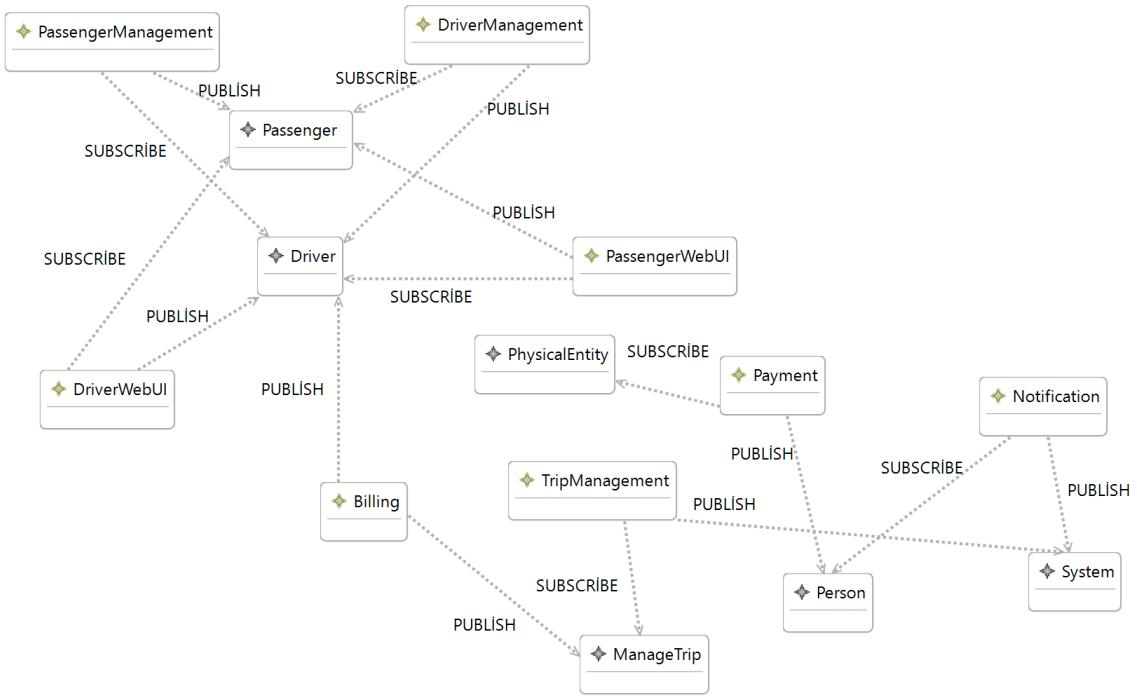
Şekil 8.6. Yayınla/Abone ol ilişkilerinin kaynak ve hedef özelliklerinin belirlenmesi

Çalışma kapsamında aşağıdaki ilişkiler kurulmaktadır:

- PassengerManagement servisi Passenger mesajını yayınlar, Driver mesajına abone olur.
- DriverManagement servisi Driver mesajını yayınlar, Passenger mesajına abone olur.
- Billing servisi ManageTrip mesajını yayınlar, Driver mesajına abone olur
- DriverWebUI servisi Driver mesajını yayınlar, Passenger mesajına abone olur.

- Notification servisi System mesajını yayınlar Person mesajına abone olur.
- TripManagement System mesajını yayınlar, ManageTrip mesajına abone olur.
- Payment servisi Person mesajını yayınlar, PhysicalEntity mesajına abone olur.

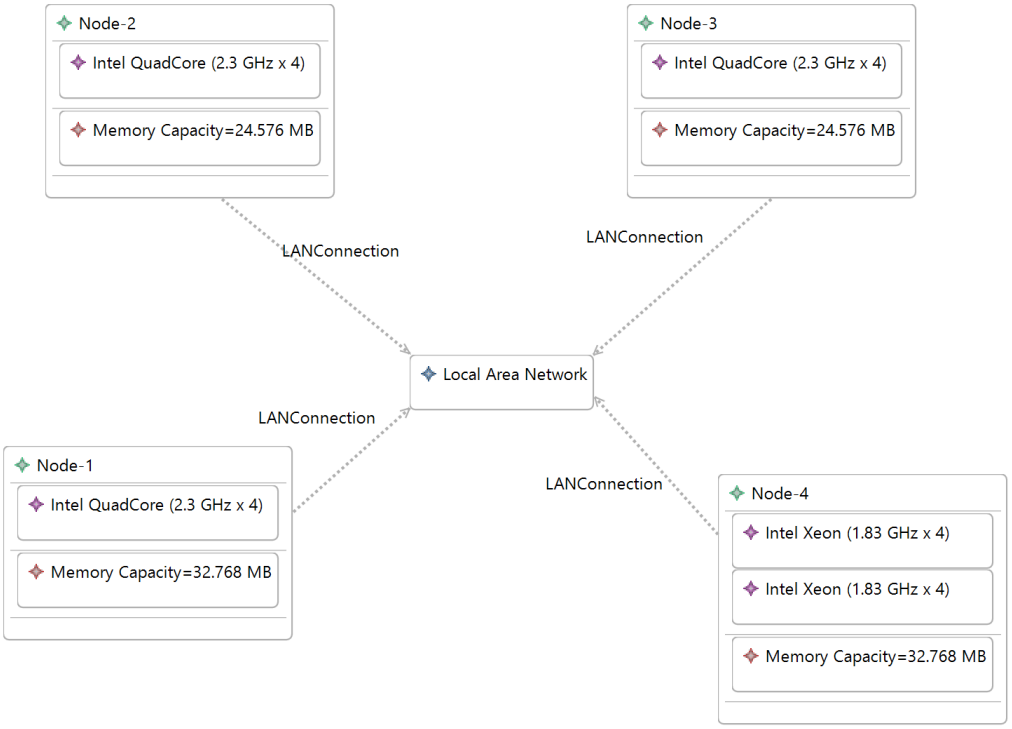
Servisler arası iletişim tanımlanırken her bir mikroservis, modelleme sorumluluğuna sahip oldukları nesneyi yayınlamaktadır. Örneğin *PassengerManagement* servisi, yolcu nesnesinden sorumlu olduğu için *Passenger* mesajını yayınlamaktadır. Aynı zamanda ilgili verilerin gerekli güncellemelerini almak için nesne sınıflarına abone olmaktadır. Örneğin *TripManagement* servisi, aracın başlangıç ve bitiş konumlarından haberdar olmak için *ManageTrip* mesajına abone olmaktadır. Şekil 8.7.'de birbirleri ile iletişimde bulunan tüm servislerin bağlantısı diyagram üzerinde görülebilmektedir.



Şekil 8.7. Taksi çağırma sistemi için oluşturulan Mikroservis İletişim Modeli

8.2.3. Mikroservislerin Dağıtılacağı Fiziksel Altyapının Oluşturulması

Mikroservislerin dağıtılması için gerekli olan fiziksel altyapı, durum çalışması kapsamında Şekil 8.8.'de görüldüğü gibi farklı işlemcilere ve farklı hafıza kapasitesine sahip 4 düğümden oluşmaktadır. Şekilde görüldüğü gibi Node-4 gibi bazı düğümler birden fazla işlemciye sahip olabilmektedir. Bu dört düğüm birbirlerine Local Area Network bağlantısı ile bağlanmaktadır.

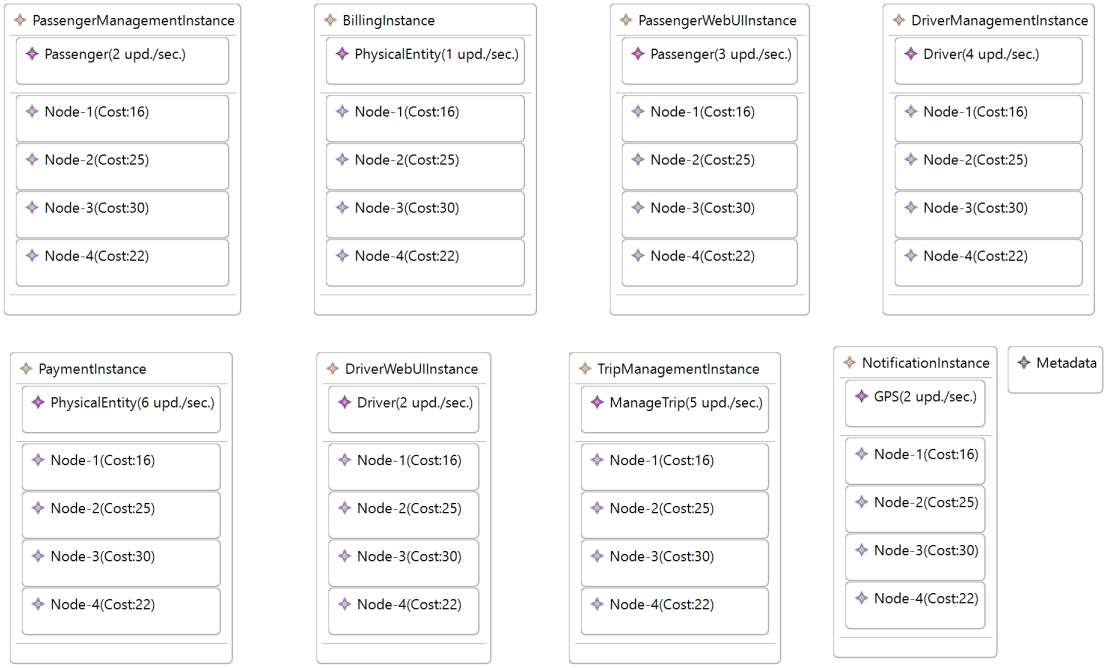


Şekil 8.8. Durum çalışması için oluşturulan dört düğümlü Mikroservis Altyapı Modeli

8.2.4. Çalışma Zamanı Yürüme Konfigürasyonu için Mikroservis Örneklerinin Tanımlanması

Mikroservis Tanımlama Modeli'nde oluşturulan mikroservislerin düğümler üzerindeki tahmini çalışma maliyeti, hangi servisten kaç adet olduğunu belirten örnek sayısı, servis örneğinin hangi mikroservis ile bağlantılı olduğu, bağlı olduğu nesne elemanına göre güncellenme

sıklığı ve dağıtılması için ihtiyaç duyulan hafıza miktarı gibi birçok özellik Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'nde tanımlanmaktadır. Taksi çağırma uygulamasının çalışma zamanı yürütme konfigürasyonu Şekil 8.9.'de görüldüğü gibidir.



Şekil 8.9. Durum çalışması için oluşturulan Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli

Bu model, mikroservislerin düğümler üzerindeki çalışma maliyetinin belirtilmesi için Mikroservis Altyapı Modeli'ni, nesneler ile bağlantı kurulabilmesi için Mikroservis Veri Değişim Modeli'ni ve hangi servisten kaç adet örnek olduğunun belirlenmesi için ise Mikroservis Tanımlama Modeli'ni kullanmaktadır. Durum çalışması için mikroservis sayılarının belirtildiği örnek senaryo Çizelge 4.1.'de verilmektedir.

8.2.5. Taksi Çağırma Uygulaması için Verimli Mikroservis Dağıtım Modellerinin Türetilmesi

Micro-IDE aracının mikroservisleri dağıtması için gerekli olan tüm fiziksel ve yapısal altyapılar kurulduktan sonra Çizelge 4.1.'de tanımlanan mikroservis örneklerinin algoritmik

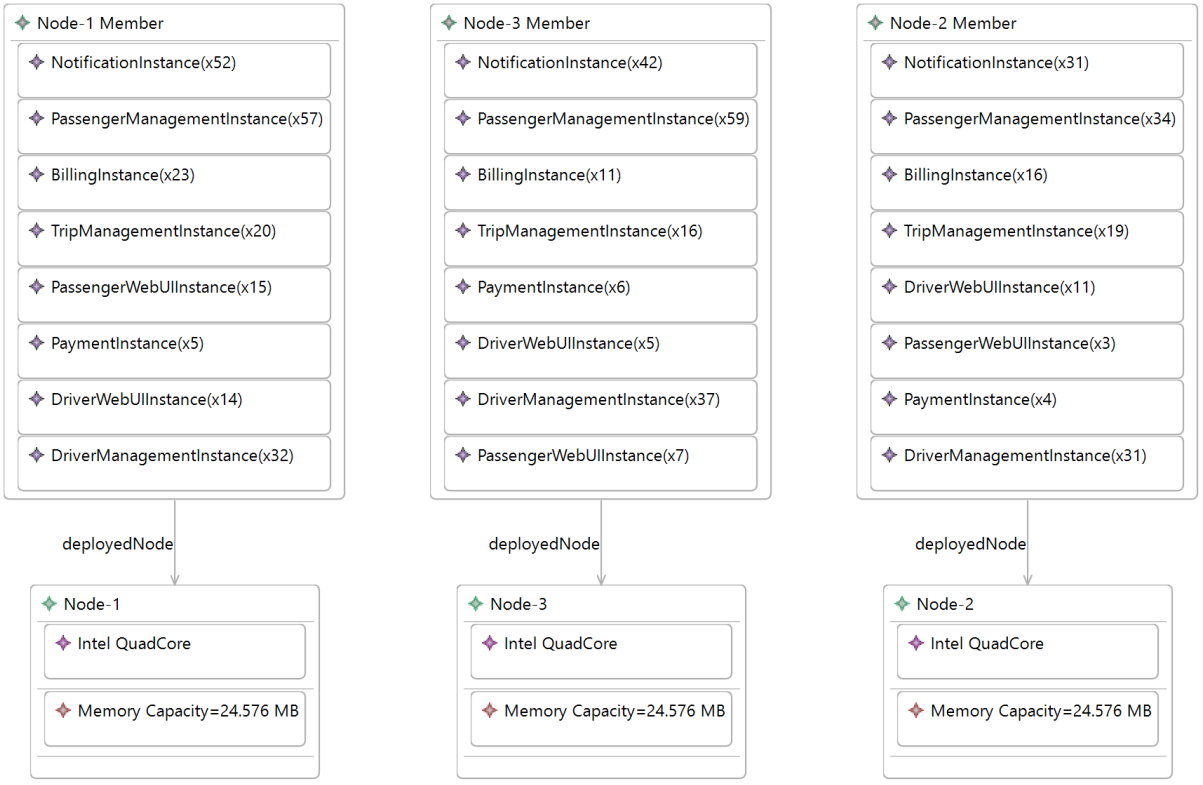
yaklaşımlar kullanılarak otomatik dağıtım işlemi gerçekleşmektedir. Bu dağıtımın gerçekleştirilmesinde kullanılan genel algoritma aşağıda verilmektedir.

```
1: GENERATE_EFFICIENT_DEPLOYMENT (phy_resources, runtime_exec_config)
2:     processors← EXTRACT_PROCESSORS (phy_resources)
3:     tasks← EXTRACT_TASKS (runtime_exec_config)
4:     allocation_table←EXECUTE_CTAP (tasks, processors)
5:     CREATE_DEPLOYMENT_MODEL (allocation_table)
```

Algoritmanın ilk satırında yer alan GENERATE_EFFICIENT_DEPLOYMENT metodu, Şekil 8.8.'de tanımlanan Mikroservis Altyapı Modeli ve Şekil 8.9.'de görülen Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'ni kullanmak üzere iki parametre içermektedir. İkinci satırda EXTRACT_PROCESSORS Mikroservis Altyapı Modeli' ndeki her düğüm için oluşturulan işlemciyi temsil etmektedir. Bu metot ile altyapı modelindeki işlemcilerin özellik çıkarımı gerçekleşmektedir. Üçüncü satırda, CTAP algoritmasında görev (task) kavramına karşılık gelen mikroservislerin düğümler üzerindeki çalışma maliyeti ve servisler arası iletişim maliyetini çıkaran EXTRACT_ TASKS metodu çağırılarak mikroservis örnekleri çalışma zamanı konfigürasyon modelinden elde edilmektedir. Dördüncü satırda ise işlemciler ve mikroservisleri parametre olarak alan EXECUTE_CTAP metodu ile gerçek CTAP algoritması uygulanmaktadır. Metotlardan alınan maliyet bilgilerine göre allocation_table isimli tabloya mikroservislerin işlemcilere atamaları kaydedilmektedir. Tablodaki allocation_table tablosunda işlemcilere ataması yapılan her bir üye uygulanabilir bir dağıtım alternatifinin soyut bir tanımını ifade etmektedir. Son satırda ise assignment_tables tablosundaki parametreler ile CREATE_DEPLOYMENT_MODEL metodu çağırılarak alternatif dağıtımlar algoritmalar kullanılarak üretilmektedir.

Bu tez kapsamında Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli üzerinde tanımlanan mikroservis örneklerinin dağıtımı için CTAP yöntemine uyarlanan birçok algoritma kullanılmaktadır. Böylece algoritmalar arasında performans değerlendirmesi yapılarak hangi algoritmada daha çok iyileşme olduğu incelenmektedir. Şekil 8.10.'da Hungarian algoritması kullanılarak eş kapasiteli tasarlanan üç düğüme servis örneklerinin atanması sonucu

elde edilen dağıtım görülmektedir.



Şekil 8.10. Mikroservis örneklerinin Hungarian algoritması ile fiziksel kaynaklara dağıtılması

8.3. Spotify Müzik ve Podcasts Uygulaması için Micro-IDE Kullanımı

Bu bölümde, Micro-IDE aracının başka bir durum çalışmasında test edilmesi için Bölüm 4.1'de tanımlanan Spotify uygulamasının küçük bir bölümü kullanılmaktadır. Bu durum çalışmasının Micro-IDE aracında uygulanması ile oluşturulan modeller ve bu modeller kullanılarak türetilen dağıtım modeli aşağıda alt başlıklar halinde açıklanmaktadır.

8.3.1. Veri Değişim Metamodelinin Tanımlanması-Nesne Modelleri ve Veri Tipleri

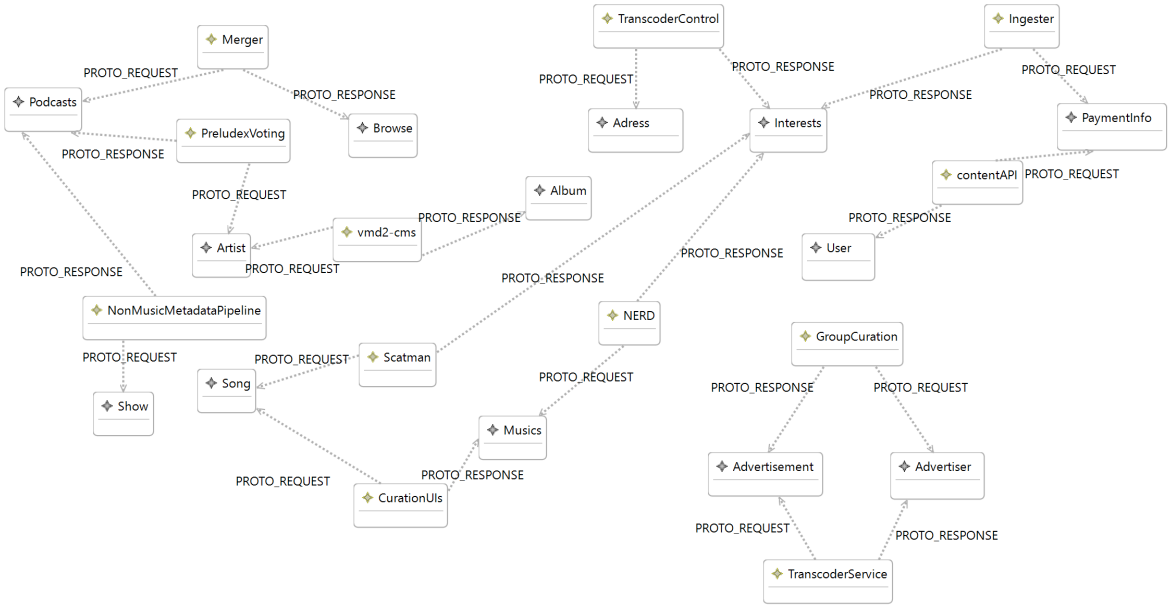
Spotify uygulamasının veri değişim modeli tasarlanırken, öncelikle bu uygulamada kullanılan nesneler belirlenmektedir. Şekil 8.11.'de oluşturulan tüm veri değişim modelinin nesne sınıfları, sınıflara ait özellikler ve bu özelliklerin yanında veri tipleri gösterilmektedir. Spotify

uygulamasında, *User* isimli bir nesne sınıfı oluşturularak *userNo*, *password*, *email* bilgileri özellik olarak bu sınıfa eklenmiştir. Kullanıcı üzerine tasarlanan veri değişim modelinde kullanıcının dinlediği müzikler, podcastler, üye ödeme bilgileri (*interests*) ve arama listesi (*browse*) isimli nesne sınıfları oluşturulmaktadır. Bu sınıflar kullanıcıya özel olduğu için *User* sınıfı kök sınıf olarak tasarlanmaktadır. *EnumeratedDataType* olarak tanımlanan *Subscribing* ve *PremiumTypeEnum* nesne sınıfları ise *PaymentInfo* sınıfında sırasıyla ödeme şeklinin ve üyelik türünün seçilmesine olanak tanımaktadır. Ödeme şekli özelliğini gösteren Subscribing sınıfı ile kredi kartı, telefon faturası veya ön ödeme yapılarak sisteme üye olunabilmektedir. Benzer şekilde *PremiumTypeEnum* nesne sınıfında yer alan *individual*, *student* veya *family* seçimi ile bir üyenin seçilen özelliğe göre ücretlendirilmesi sağlanmaktadır. Ayrıca kök sınıfı *Musics* olan *Artist*, *Album*, *Song* sınıflarında ise *name*, *genre*, *rating* ve *listenCount* özellikleri yer almaktadır.

Şekil 8.11. Spotify müzik uygulaması için tasarlanan Mikroservis Veri Değişim Modeli

8.3.2. Mikroservislerin Belirlenmesi ve Aralarındaki İletişim Deseninin Kurulması

Bu durum çalışması kapsamında Çizelge 4.2.' de isimleri verilen 18 adet mikroservis tanımlanarak servislerin birbirleri ile haberleşmesinde gRPC iletişim deseni kullanılmaktadır. Oluşturulan mikroservisler üzerinde istenilen iletişim deseni seçildikten sonra ilişkiye ait özellikler kısmından gRPC tipi ile hangi servisin hangi veriye istekte bulunduğu (*ProtoRequest*) ve hangi veriyi cevap olarak gönderdiği (*ProtoResponse*), ilişkinin hangi mikroservise ait olduğu (source) ve servisin hangi veri ile ilişkili olduğu belirlenebilmektedir. Şekil 8.12.'de birbirleri ile iletişimde bulunan servisler görülebilmektedir.



Şekil 8.12. Spotify müzik uygulaması için oluşturulan Mikroservis İletişim Modeli

8.3.3. Mikroservislerin Dağıtımı için Fiziksel Altyapının Kurulması

Mikroservislerin dağıtılması için gerekli olan fiziksel altyapı, Infrastructure Metamodeli'nin çalıştırılması sonucunda kurulabilmekte ve bu durum çalışması kapsamında Şekil 8.8.'de görülen fiziksel altyapı modeli kullanılmaktadır. Önerilen araç istenilen sayıda ve özellikte düğümü oluşturmaya ve heterojen LAN/WAN bağlantısı oluşturmaya olanak tanımaktadır.

8.3.4. Çalışma Zamanı Yürütme Konfigürasyon Metamodelini Kullanarak Mikroservis Örneklerinin Tanımlanması

Mikroservis Tanımlama Modeli'nde oluşturulan 18 adet mikroservisten hangi servis örneğinden kaç adet olduğunu belirten örnek sayısı (*instanceCount*), servis örneğinin hangi mikroservis ile bağlantılı olduğu (*relatedMicroservice*) ve dağıtılması için ihtiyaç duyulan hafıza miktarı gibi birçok özellik Çalışma Zamanı Yürütme Konfigürasyon Modeli'nde tanımlanmaktadır. Ayrıca bir mikroservis örneğinde *Publication* ve *ExecutionCost* sınıfları özellik olarak eklenerek servis örneğinin bağlı olduğu veriyi güncelleme sıklığı (*updateRate*) ve her bir düğüm üzerinde çalışma maliyeti belirlenmektedir. Spotify uygulaması için tasarlanan örnek Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli'nin bir kısmı Şekil 8.13.'de verilmektedir. 18 adet mikroservis için çoklu mikroservis örneği nesnesi tanımlanmaktadır.

Şekil 8.13. Spotify müzik uygulaması için oluşturulan Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modelinin bir bölümü

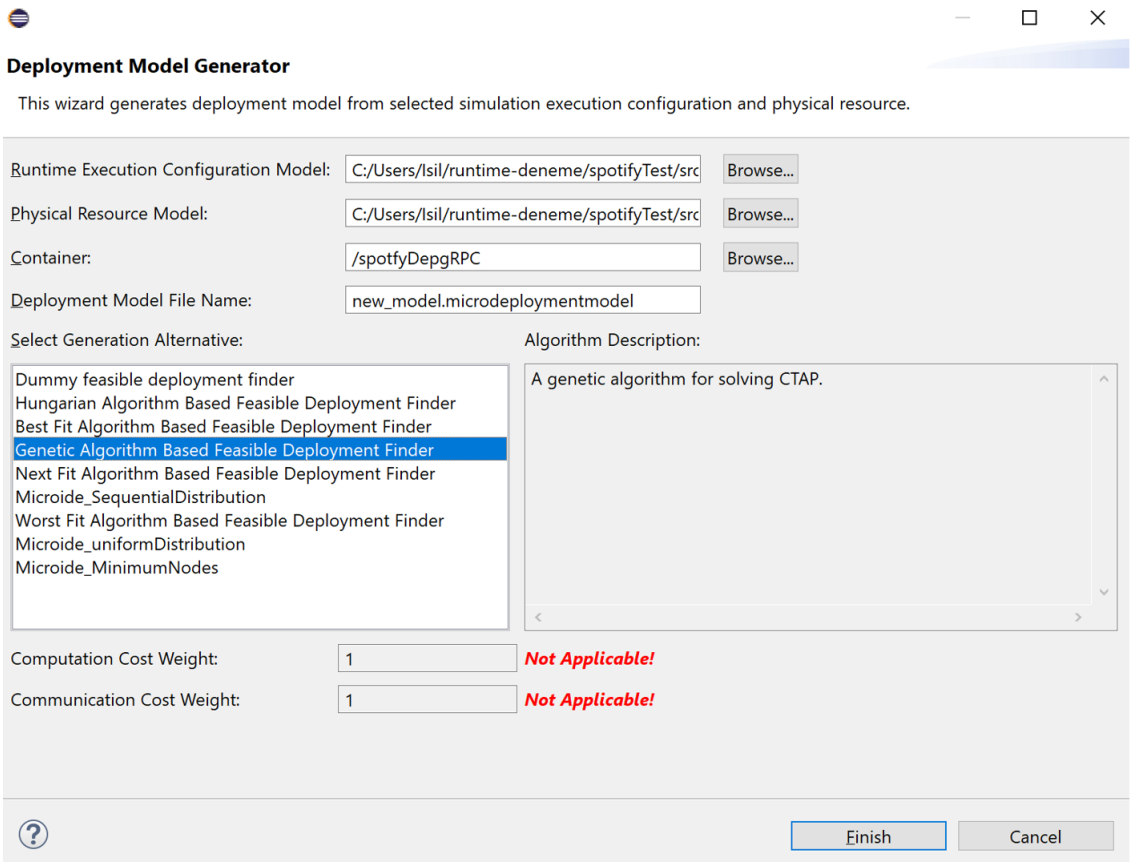
Bu model, mikroservislerin düğümler üzerindeki çalışma maliyetinin belirtilmesi için Mikroservis Altyapı Modeli'ni, nesneler ile bağlantı kurulabilmesi için Mikroservis Veri Değişim Modeli'ni ve hangi servisten kaç adet örnek olduğunun belirlenmesi için ise Mikroservis Tanımlama Modeli'ni kullanmaktadır. Spotify durum çalışması için oluşturulan bu Mikroservis Altyapı Modeli Çizelge 4.2.'de belirtilen mikroservis sayılarını kullanmaktadır.

9. DEĞERLENDİRME

Bu bölümde Micro-IDE aracının kullanımı sonucu elde edilen dağıtım modellerinin uygunluğu ve dağıtım alternatifi üretimi esnasında algoritmalara göre değişen zaman performansı tartışılmaktadır.

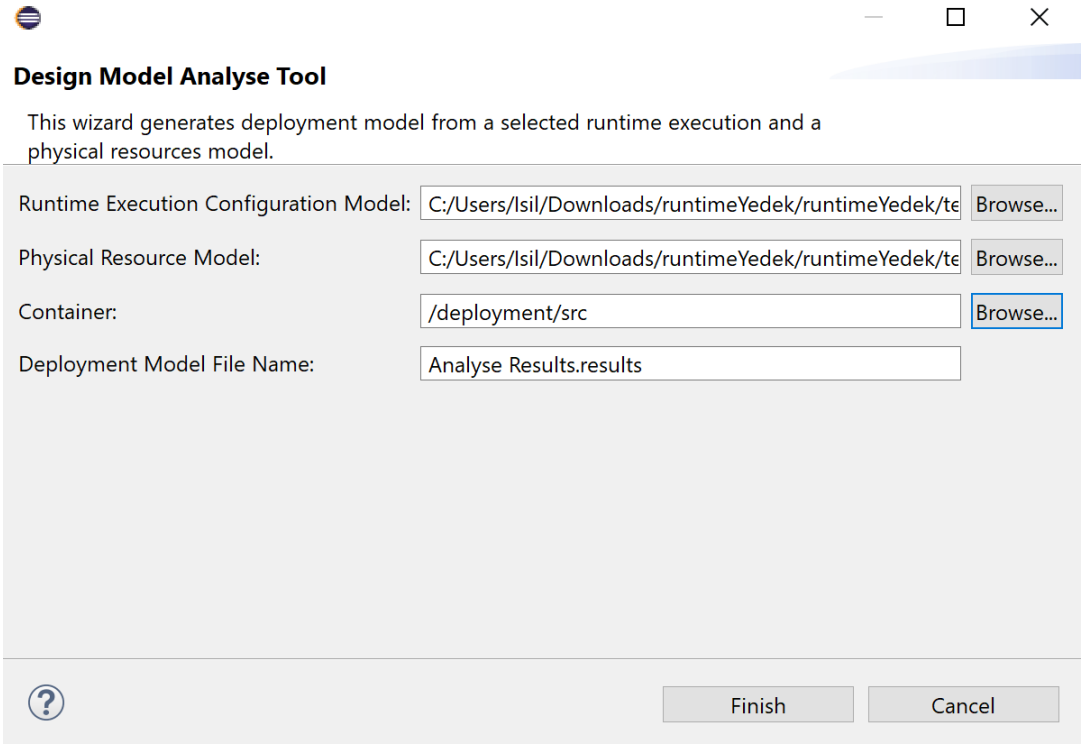
9.1. Uygulanan Algoritmaların Performans Değerlendirmesi

Bölüm 8. başlığında anlatılan modeller kullanıcı tarafından tasarlandıktan sonra araç üzerinde seçilen çalışma zamanı yürütme konfigürasyon modeli ve fiziksel altyapılar ile birlikte hangi algoritmanın uygulanacağı da Şekil 9.1.'de görüldüğü gibi seçilebilmektedir.



Şekil 9.1. Micro-IDE aracının model üretme arayüzü

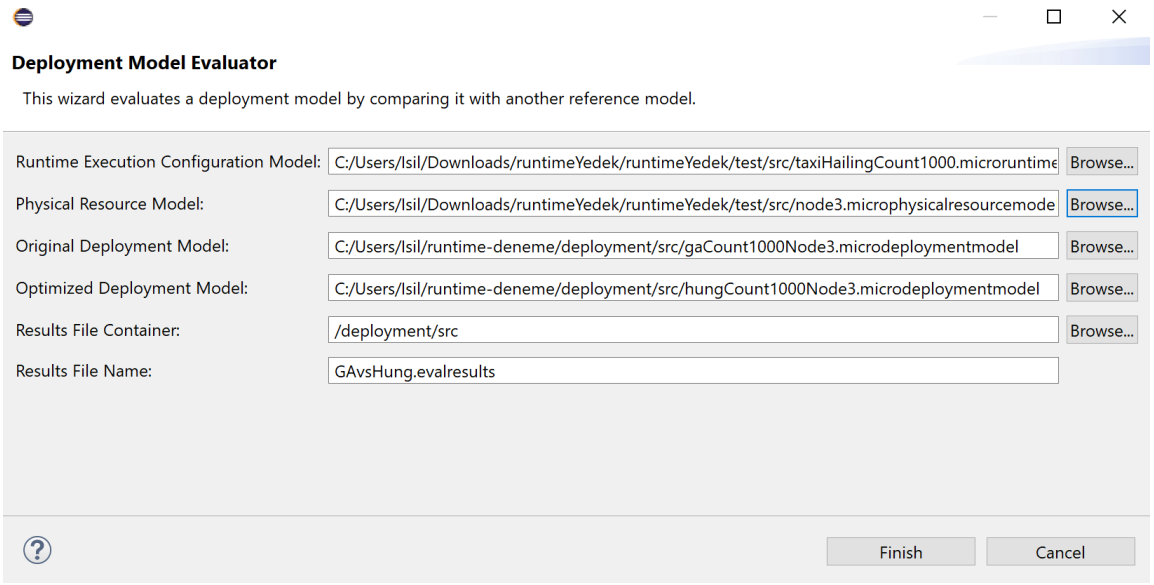
Bununla birlikte; tasarlanan iletişim modeli, veri değişim modeli, fiziksel altyapı modeli ve mikroservis örneği sayılarına göre tasarım modelinin analiz edildiği bir arayüz de Micro-IDE aracında geliştirilmiştir. Şekil 9.2.'de verilen arayüz üzerinden Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli ve Mikroservis Altyapı Modeli seçilerek .results dosyası elde edilmektedir. Bu dosya, mikroservislerin hafıza gereksinimleri, kullandığı nesnelerin boyutları, mikroservisler arasındaki iletişim maliyetleri ve araç üzerinde seçilen fiziksel kaynakların hafıza kapasiteleri hakkında kullanıcıya ham bilgi sunmaktadır.



Şekil 9.2. Micro-IDE aracının model analizi arayüzü

Analiz arayüzünden elde edilen bilgilerde öncelikle mikroservislerin iletişim için kullandığı nesnelerin boyutları listelenerek servis örneklerinin toplam iletişim maliyetleri, örnek sayıları ve nesne boyutlarının çarpımı ile belirlenmektedir. Bununla birlikte, kullanılan fiziksel kaynakların hafıza kapasiteleri hakkında bilgi verilerek her bir mikroservis örneğinin dağıtımı için gerekli hafıza gereksinimleri listelenmektedir.

Micro-IDE aracının üçüncü arayüzü ise elde edilen dağıtım modellerinin birbirleri ile karşılaştırılması için veya manuel dağıtım yaklaşımı sonucu elde edilen dağıtım modeli ile algoritmik yaklaşımlar kullanılarak üretilen dağıtım modelinin karşılaştırılması için oluşturulan bir arayüzdür. Şekil 9.3.'de görüldüğü gibi Original ve Optimized olmak üzere iki farklı dağıtım modeli, seçilen çalışma zamanı yürütme konfigürasyon modeli ve fiziksel altyapı modeli üzerinden karşılaştırılmaktadır.



Şekil 9.3. Micro-IDE aracının modelinin değerlendirme arayüzü

Bu çalışmada algoritmaların performans degerlendirmesinin yapılması için her bir algoritma için 1361 adet mikroservis örneginin 4 düğümden oluşan farklı kapasitelere sahip işlemcilere dağıtılması işlemi yapılmaktadır. Hungarian, Genetik, Next Fit, Best Fit, Worse Fit, Sequential Distribution, Uniform Distribution ve Minimum Node algoritmaları olmak üzere 8 çeşit algoritma Micro-IDE aracında uygulanmaktadır. Genetik algoritmanın bir çok çeşidi olmakla birlikte çalışmada CTAP algoritmasına özgü Mehrabi et. al [94] tarafından önerilen genetik algoritma kullanılmaktadır. Bu algoritma görevler arası iletişim maliyetini de hesaba katarak CTAP'e optimal bir yaklaşım sunmaya çalışmaktadır.

Her bir algoritma, $ExecutionCost(relativeunit)$ ve $CommunicationCost(Mbytes/s)$ metriklerini kullanarak bir veya birden fazla farklı dağıtım alternatifi sunabilmektedir. Mimaride

yer alan mikroservisler arası toplam iletişim maliyeti $CommunicationCost$ ifadesi ile tanımlamaktadır. $ExecutionCost$ ise mimaride kullanılan düğümlerin üzerinde çalışan mikroservislerin toplam çalışma maliyetini ifade etmektedir.

Çalışma kapsamında öncelikle Micro-IDE aracının en verimli dağıtım alternatifleri üretirken geçen zamanın makul olup olmadığı belirlenmektedir.

Algoritmanın dağıtım modeli üretim süreleri incelendiğinde az sayıda servisin bir saniyeden az bir sürede dağıtılabildiği, servis sayısı arttıkça sürenin doğru orantılı olarak uzadığı görülmektedir. Fakat fazla sayıda servisin dağıtımında bile elde edilen sürelerin kabul edilebilir olduğu ve dağıtımların bir manuel olarak yapılan bir dağıtıma göre daha hızlı sonuç verdiği görülmektedir.

İkinci bir değerlendirme olarak algoritmaların birbirlerine göre başarım oranlarını incelemek adına kullanılan her bir algoritmanın bir optimizasyon algoritması olan genetik algoritmaya göre toplam iletişim maliyeti ve toplam çalışma maliyetinin performans iyileşiminin yüzdelik oranı karşılaştırılmaktadır. Micro-IDE aracını test etmek için bu tez kapsamında iki adet durum çalışması üzerinden algoritmaların performans ve zaman değerlendirmesi yapılmaktadır.

9.2. Taksi Çağırma Sistemi (TÇS) için Algoritmaların Performans Değerlendirmesi

Bu bölümde ilk durum çalışması olarak belirlenen TÇS için Bölüm 8.'de tasarımı verilen metamodeller kullanılarak ilk olarak manuel dağıtım ve genetik algoritma karşılaştırılmakta, ikinci olarak da Micro-IDE aracında uygulanan tüm algoritmaların genetik algoritmaya göre performans karşılaştırılması yapılmaktadır. Genetik algoritma ve manuel dağıtım ile elde edilen dağıtım modellerinin servis başına düşen iletişim maliyetleri, servislerin düğümler üzerindeki toplam çalışma maliyetleri ve manuel dağıtıma göre algoritmik yaklaşım ile yapılan dağıtımın iyileşme oranları Çizelge 9.1.'de listelenmektedir.

Çalışma kapsamında kullanılan iletişim maliyeti, Mikroservis İletişim Modeli üzerinde tanımlı mikroservisler arasında oluşturulan toplam iletişim maliyetini ifade etmektedir. Toplam iletişim maliyeti, servislerin kullandığı veri değişim nesneleri ve bu verileri ne kadar sürede güncelledikleri göz önüne alınarak hesaplanmaktadır. Çalışma maliyeti ise düğümlere atanan mikroservislerin bağlı bulunduğu düğüm üzerindeki toplam çalışma maliyetini temsil etmektedir. Oran sütununda belirtilen yüzdelik ise manuel dağıtıma oran ile genetik algoritmanın her iki maliyet açısından da iyileşme yüzdelerini ifade etmektedir.

Çizelge 9.1. Manuel olarak ve genetik algoritma ile yapılan dağıtım modellerinde servis başına düşen iletişim ve çalışma maliyetlerinin karşılaştırılması

Mikroservis örneği	Çalışma maliyeti (birim)			İletişim maliyeti (Mbytes/sn)		
	Manuel dağıtım	Genetik algoritma	Oran (%)	Manuel dağıtım	Genetik algoritma	Oran (%)
DriverManagementIns. (x100)	2325	2306	0.81	2.265	2.234	1.36
DriverWebUIInstance (x30)	695	660	5.03	0.293	0.292	0.34
PaymentInstance (x15)	350	319	8.85	0	0	0
BillingInstance (x50)	1160	1172	-1.03	0.00645	0.00647	-0.3
PassengerManagementIns. (x150)	3485	3455	0.86	1.4019	1.4256	-1.69
NotificationInstance (x125)	2905	2851	1.85	0	0	0
TripManagementInstance (x55)	1275	1270	0.39	0	0	0
PassengerWebUIInstance (x25)	705	600	13.4	0.3604	0.3031	15.89
Toplam	**12.900**	**12.633**	**2.069**	**4.3281**	**4.2626**	**1.51**

Genetik algoritma ile üretilen dağıtım modeli ile manuel dağıtım modeli karşılaştırıldığında toplam iletişim ve çalışma maliyetleri açısından manuel dağıtımın nispeten daha düşük performans gösterdiği Çizelge 9.1.'nun son satırındaki iyileşme oranlarından görülmektedir.

İletişim maliyetleri değerlendirildiğinde, *PaymentInstance*, *NotificationInstance*, ve *TripManagementInstance* isimli servisler arasındaki maliyetleri sıfır olduğu görülmektedir. Bunun sebebi veri alışverişinde bulunan her iki servis arasında tek bir maliyet hesaplandığı için bu değer sadece yayıncı üzerinde gösterilmektedir. Örneğin, bu çalışma kapsamında TÇS için oluşturulan senaryoda *TripManagementInstance* servisi *ManageTrip* nesnesine (yolculuk konumunun başlangıç ve bitişini temsil eden veri) üye olurken *BillingInstance ManageTrip* nesnesini yayınlamaktadır. Bu yüzden *TripManagementInstance* ve *BillingInstance* arasındaki iletişim maliyeti yalnızca yayıncı (*BillingInstance*) tarafında gösterilmektedir.

Manuel dağıtım sonucunda mikroservis bazında hesaplanan iletişim ve çalışma maliyetlerinden bazıları genetik algoritmadan daha iyi sonuç verirken önerilen bu yaklaşımda amaç algoritmik olarak sistemin toplam iletişim ve çalışma maliyetini minimuma yakın bir sonuca ulaştırmak ve birden fazla verimli dağıtım alternatifini tasarımcıya sunmaktır. Toplam maliyet açısından değerlendirme yapıldığında, genetik algoritma kullanılarak %2.06 oranında toplam çalışma maliyeti, %1.51 oranında ise toplam iletişim maliyeti açısından iyileşme olduğu görülmektedir.

İyileştirme oranları tasarlanan mimariye, durum çalışmasına ve seçilen CTAP algoritmasına göre değişmektedir. Önerilen bu yaklaşım ve araç desteği ile amaç, sistemi iyi bilen bir uzmanın manuel olarak dağıtım yapmasına yardımcı olabilecek verimli dağıtım alternatifleri sunmak ve manuel dağıtıma yakın veya daha iyi alternatifler elde etmektir. Mikroservis tabanlı bir mimaride servis sayısı arttıkça manuel olarak yapılan dağıtımda verimli alternatiflerin oluşturulması zorlaşmaktadır. Bu sebepten dolayı önerilen algoritmik dağıtım yaklaşımı manuel dağıtıma göre daha kısa sürede ve daha fazla verimli dağıtım alternatifi sunarak avantajlı bir yaklaşım haline gelmektedir.

İkinci değerlendirme kriteri olarak tüm algoritmaların TÇS durum çalışması üzerinde performans değerlendirmesinin yapılması için her bir algoritma üzerinden 1000 adet mikroservis örneğinin 3 düğümden oluşan eş kapasitelere sahip (25.576 MB) işlemcilere dağıtılması işlemi yapılmaktadır. Hungarian, Genetik [94], Next Fit, Sequential Distribution, Uniform Distribution ve Minimum Node Distribution algoritmaları olmak üzere 6 adet algoritma Micro-IDE aracına uygulanmaktadır. Bu algoritmalar mikroservisler arası iletişim maliyetini de hesaba katarak CTAP problemine optimal bir yaklaşım sunmaya çalışmaktadır.

Algoritmaların birbirlerine göre başarım oranlarını incelemek adına kullanılan her bir algoritmanın bir optimizasyon algoritması olan genetik algoritmaya göre toplam iletişim maliyeti ve toplam çalışma maliyetinin performans iyileşiminin yüzdelik oranı karşılaştırılmakta ve değerlendirme sonuçları Çizelge 9.2.'de görülmektedir.

Çizelge 9.2. Algoritmaların genetik algoritmaya göre performans karşılaştırması (GA: Genetik Algoritma, MD: Minimum Distribution, SD: Sequential Distribution, UD: Uniform Distribution, HA: Hungarian, NF: Next Fit, BF: Best Fit, WF: Worst Fit)

Algoritma	Toplam iletişim maliyeti	İyileşme oranı	Toplam çalışma maliyeti	İyileşme oranı
HA	7.9828644	% 1.36	22723	% 3.17
GA	8.09317	-	23466	-
NF	2.3872528	% 70.5	17863	% 23.8
BF	2.3872528	% 70.5	17863	% 23.8
WF	3.8068085	% -0.19	13018	% 0.55
SD	8.108475	% -0.18	23659	% -0.82
UD	8.108269	% -0.18	23659	% -0.82
MD	4.374584	% 45.9	18952	% 19.2

Çizelge 9.2.'te listelenen sonuçlara göre Next Fit ve Best Fit algoritmaları kullanılarak elde edilen toplam iletişim ve çalışma maliyetindeki iyileşme oranlarının (sırasıyla %70 ve %23.8)

genetik algoritmaya göre oldukça yüksek olduğu görülmektedir. Minimum Nodes algoritması iletişim ve çalışma maliyeti açısından %45.9 ve %19.2 oranları ile genetik algoritmaya göre daha iyi sonuç vermektedir. Bunun sebebi ise bu algoritmalar, yük dengelemeye bakmaksızın düğümün yeterli hafızası var ise alabileceği kadar servisi düğüme dağıtmaktadır. Bu durumda iletişim maliyetleri göz önüne alınmadan yalnızca hafıza kapasitesine göre dağıtım yaptıkları için genetik algoritmaya göre daha az düğüm kullanmaktadırlar. Bu durumda kullanılmayan düğümler üzerindeki toplam çalışma maliyeti 0 olmaktadır. Benzer şekilde aynı düğüme atanmış servisler arasındaki iletişim maliyeti de 0 olmaktadır. Bu durum her ne kadar istenen bir durum olsa da bir düğümün üzerindeki yük ne kadar fazla ise o düğümün işlev görmesi o kadar zorlaşmaktadır. Hungarian algoritması ise Genetik algoritmaya göre toplam iletişim maliyeti açısından %1.36 oranında iyileşme, toplam çalışma maliyeti açısından ise %3.17 oranında iyileşme göstermektedir. Diğer algoritmaların performansı incelendiğinde Sequential Distribution ve Uniform Distribution algoritmaları toplam iletişim ve çalışma maliyetleri açısından eşit oranlarda genetik algoritmaya göre daha düşük performans göstermektedir.

9.3. Spotify Uygulaması için Algoritmaların Performans Değerlendirmesi

Bu bölümde ikinci durum çalışması olarak belirlenen Spotify uygulaması için Bölüm 8.'de tasarımı verilen metamodeller kullanılarak tüm algoritmaların ürettiği dağıtım modellerinin toplam iletişim maliyeti ve düğümler üzerindeki toplam çalışma maliyeti değerlendirilmektedir. Hem algoritmik yaklaşımlar sonucu üretilen dağıtım modelleri hem de manuel olarak oluşturulan dağıtım modelinin değerlendirme sonuçları sırası ile Çizelge 9.3.'de ve Çizelge 9.4.'de görülmektedir.

Çizelge 9.3. Seçilen algoritma için toplam iletişim ve çalışma maliyetleri (GA: Genetik Algoritma, MD: Minimum Distribution, SD: Sequential Distribution, UD: Uniform Distribution, HA: Hungarian, NF: Next Fit, BF: Best Fit, WF: Worst Fit)

Algoritma	Toplam iletişim maliyeti (Mbytes/sn)	Düğümler üzerindeki toplam çalışma maliyeti (birim)
HA	268.122	21.425
GA	269.327	21.603
NF	196.922	**18.918**
BF	217.109	22.390
WF	263.391	21.881
SD	269.735	21.772
UD	254.489	21.844
MD	**172.824**	21.855

Çizelge 9.4. Manuel olarak yapılan dağıtım modelinden elde edilen toplam iletişim ve çalışma maliyetleri

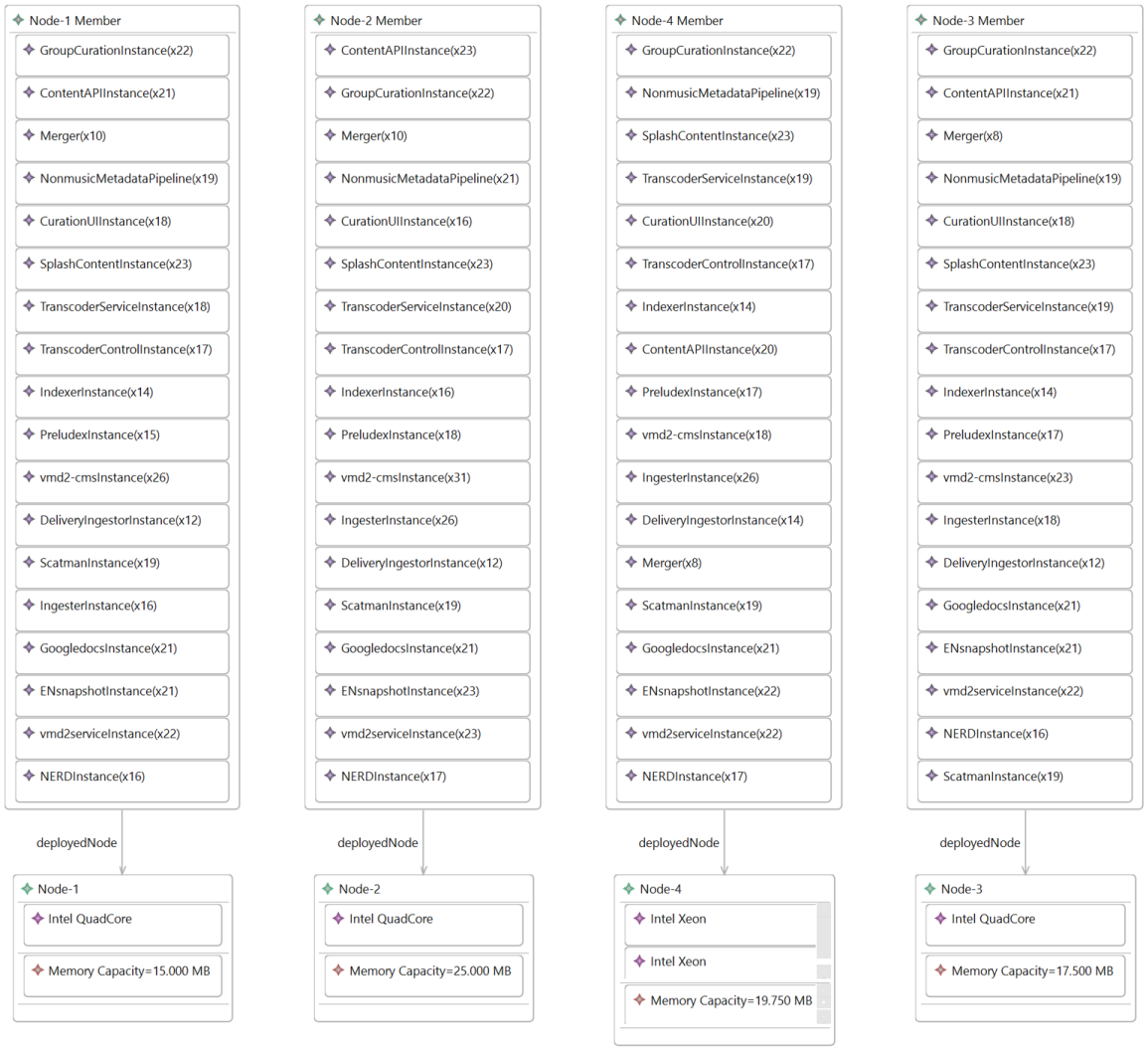
Dağıtım yaklaşımı	Toplam iletişim maliyeti (Mbytes/sn)	Toplam çalışma maliyeti (birim)
Manuel dağıtım	270.879	21.676

Çizelge 9.3.'de gözlemlenen değerlere göre en düşük toplam iletişim maliyeti Minimum Nodes algoritmasında görülürken bu algoritma çalışma maliyeti açısından en yüksek değerlerden birini vermektedir. Çalışma maliyeti açısından tüm algoritmalar içerisinden en düşük maliyet Next Fit algoritmasında görülmektedir. Her iki metrik açısından değerlendirme yapıldığında Next Fit algoritması çoğunlukla diğer algoritmalardan daha iyi performans göstermektedir. Tüm bu değerlerin daha detaylı analizi (her mikroservis ve her bir düğüm üzerindeki toplam iletişim ve çalışma maliyetleri, düğümler üzerinde dağıtılan servislerin toplam hafıza kapasitesi) Micro-IDE aracı ile tasarımcıya iletilmekte ve böylece araç, verimli dağıtım alternatifleri üzerinden karar vermek için tasarımcıya yararlı bilgiler sunmaktadır.

Micro-IDE aracı üzerinde üretilen dağıtım modellerinin geçerliliğini analiz ederken iki yaklaşım kullanılaktadır. İlk yaklaşım, mimariyi çok iyi bilen bir uzman tarafından dağıtım alternatiflerinin sezgisel olarak incelenmesidir. Bu yaklaşımda alternatifin üretimi otomatik olarak yapılırken mimariye en verimli dağıtım alternatifi uzman tarafından mantıksal akıl yürütme yolu ile belirlenmektedir.

İkinci yaklaşım ise bir dağıtım alternatifinin başka bir dağıtım alternatifi ile iletişim ve çalışma maliyeti ve atanan servislerin düğümlerdeki toplam kapasiteleri açısından karşılaştırılmasıdır. Micro-IDE aracının Şekil 9.3.'de Deployment Model Evaluator arayüzünde görüldüğü gibi daha önce üretilen iki dağıtım modeli, gerekli Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli (Microservice Runtime Execution Configuration Model) ve Mikroservis Altyapı Modeli (Microservice Infrastructure Model) seçildikten sonra karşılaştırılabilmektedir. Sonuçların yazıldığı dosyanın yolu ve dosya adı belirtildikten sonra Finish butonu, iki modelin karşılaştırılmasının yapıldığı sonuç dosyasını çıktı olarak vermektedir.

Önerilen araç, algoritmaların otomatik olarak ürettiği alternatiflerin yanısıra manuel olarak dağıtım alternatifi oluşturmaya da olanak tanımaktadır. Böylece bir algoritmanın ürettiği dağıtım alternatifi ile manuel olarak üretilen dağıtım modeli performans açısından karşılaştırılabilmektedir. Şekil 9.4.'de 1361 adet mikroservisin 4 düğüme manuel olarak atandığı dağıtım modeli gösterilmektedir. Üretilen dağıtım modellerinde mikroservis isimlerinin yanında parantez içerisinde servis örneğinin sayısı belirtilmektedir. Parantez içerisinde belirtilmeyen servisler ise tek adet olarak atanmaktadır.



Şekil 9.4. Manuel olarak oluşturulan dağıtım modeli

Çizelge 9.3.'de algoritmalara göre otomatik olarak oluşturulan dağıtım modellerinden elde edilen maliyetler ile Çizelge 9.4.'de manuel dağıtım ile elde edilen maliyetler karşılaştırıldığında iletişim maliyeti açısından en düşük performansın manuel dağıtıma ait olduğu görülmektedir. Çalışma maliyetleri açısından bakıldığında ise Best Fit, Worst Fit, Sequential Distribution, Uniform Distribution ve Minimum Nodes algoritmalarından az bir farkla daha başarılı olduğu görülmektedir. Fakat her iki maliyet incelendiğinde Next Fit, Genetic ve Hungarian algoritmalarının manuel dağıtımı geride bıraktığı görülmektedir. İletişim maliyetini en küçüklemedeki motivasyon, aralarında veri alışverişi fazla olan iki servisin

aynı düğüme dağıtılmasıdır. Spotify örneğinde *DeliveryIngestorInstance* tarafından gönderilen veri *ContentAPIInstance* tarafından alınmaktadır. Benzer şekilde *ScatmanInstance* ile *CurationUIInstance* servisleri aynı düğümde konumlandırılmaktadır. Buna benzer mimari içerisinde bir verinin bir mikroservis tarafından gönderilip başka bir mikroservis tarafından kullanıldığı bir çok örnek bulunmaktadır. Birbirlerine bağlı olan servislerin aynı düğüme atanması durumunda aralarındaki iletişim maliyeti sıfır olarak hesaplanmaktadır. Bu bakımdan Minimum Nodes algoritması, hali hazırda en az düğümü meşgul etmeyi hedeflediği için en düşük iletişim maliyetine sahip olmaktadır. Micro-IDE aracında uygulanan tüm algoritmaların manuel dağıtım yaklaşımına göre iletişim ve çalışma maliyeti açısından iyileşme yüzdeleri Çizelge 9.5.'te gösterilmektedir.

Çizelge 9.5. Manuel dağıtım yaklaşımına göre algoritmaların performans karşılaştırması

Algoritma	İletişim maliyetinin iyileşme oranı(%)	Çalışma maliyetinin iyileşme oranı (%)
Hungarian	1.01	1.15
Genetic	0.57	0.33
Next Fit	**27.3**	**12.72**
Best Fit	19.85	-3.29
Worst Fit	2.76	-0.94
Sequential Distribution	0.42	-0.44
Uniform Distribution	6.05	-0.77
Minimum Nodes	**36.1**	-0.82

Çizelge 9.5.'te görüldüğü üzere kullanılan algoritmik yaklaşımlar genel olarak manuel dağıtım yaklaşımına göre daha iyi sonuçlar vermektedir. Her iki metrik açısından en fazla iyileşme oranı Next Fit algoritmasında görülmektedir. İletişim maliyeti açısından incelendiğinde ise %36.1 iyileşme oranı ile Minimum Nodes algoritması en iyi performansı göstermektedir. Test edilen bu üreticilerin sonuçları farklı durum çalışmaları, farklı çalışma zamanı

yürütme konfigürasyon modelleri ve fiziksel altyapı modellerine göre farklılık gösterebilmektedir. Önerilen araç tasarımcıya en verimli alternatifi seçme imkanı sunmaktadır. Ayrıca ihtiyaç doğrultusunda araca kolay bir şekilde yeni üretim algoritmaları da eklenebilmektedir.

9.4. Algoritmalara Dayalı Dağıtım Modeli Üretme Süresinin TÇS ve Spotify Durum Çalışmaları Üzerinden Değerlendirilmesi

Çalışma kapsamında Micro-IDE aracının en verimli dağıtım alternatiflerini üretirken geçen zamanın makul olup olmadıgı belirlenmektedir. Dağıtım modeli üretme sürecinde seçilen CTAP algoritmasının performansı büyük ölçüde dağıtım süresini etkilemektedir. Aracı test etmek için kullanılan üretim algoritmaları içerisinden karmaşıklıgı en fazla olan algoritma Mehrabi et al. [94] tarafından önerilen genetik algoritmadır. Modele entegre edilen tüm algoritmalar Java programlama dili ile yazılmış olup Intel Core I-7 2.70 GHz 8 GB RAM'a sahip 64-bit bilgisayar üzerinde çalıştırılmaktadır.

Çizelge 9.6. Genetik algoritma kullanılarak servis ve düğüm sayılarına göre dağıtım modeli üretme süreleri

Üretim no	Durum çalışması	Toplam servis örneği sayısı	Düğüm sayısı	Üretim süresi (saniye)
1.	Spotify	18	4	2
2.	Spotify	53	4	2
3.	Spotify	206	4	17
4.	Spotify	274	5	15
5.	Spotify	1361	4	1014
6.	Spotify	1361	10	124
7.	Taxi-hailing system	5	3	0
8.	Taxi-hailing system	50	4	3
9.	Taxi-hailing system	128	4	15
10.	Taxi-hailing system	300	5	104
11.	Taxi-hailing system	540	4	147
12.	Taxi-hailing system	1361	4	2492
13.	Taxi-hailing system	1361	10	1040

Çizelge 9.6.'da genetik algoritma kullanılarak farklı sayıda servis ve düğümler ile üretilen dağıtım modelinin süresi belirtilmektedir. Servis ve düğüm sayıları endüstriyel perspektif açısından belirlenerek gerçekçi bir yaklaşım sergilenmektedir. Ele alınan iki farklı durum çalışması arasından Taxi-hailing sistemi iletişim desenleri açısından daha karmaşık bir yapıda tasarlandığı için dağıtım modeli üretim süreleri de Spotify durum çalışmasına göre daha fazladır. Yapılan analizler sonucunda mikroservislerin düğümler üzerindeki çalışma maliyetleri eşit olarak alındığında ve düğüm sayısı artırıldığında genetik algoritmanın daha kısa sürede dağıtım alternatifi ürettiği görülmektedir. Üretilen bu sonuçlara bakılarak fiziksel kaynak sayısı arttıkça aynı sayıda servisin daha kısa sürede dağıtım yapabildiği söylenebilmektedir.

Algoritmanın dağıtım alternatiflerini üretim süreleri genel olarak incelendiğinde az sayıda

servisin bir saniyeden az bir sürede dağıtılabildigi, servis sayısı arttıkça sürenin doğru orantılı olarak uzadığı görülmektedir. Fakat çok sayıda servisin dağıtımında bile elde edilen sürelerin kabul edilebilir olduğu ve dağıtımların bir manuel dağıtım yaklaşımına göre çok daha hızlı sonuç verdiği görülmektedir. Farklı CTAP algoritmalar farklı üretim sürelerinde performans göstermektedir. Araç için kullanılan tüm algoritmaların üretim süreleri, en fazla servis sayısının en az düğüme atandığı Spotify durum çalışması üzerinden Çizelge 9.7.'te karşılaştırılmaktadır.

Çizelge 9.7. Aynı sayıda servis ve düğüm sayılarına göre algoritmaların ve manuel yaklaşımın dağıtım modeli üretme süreleri (Toplam mikroservis örneği sayısı: 1361, toplam düğüm sayısı: 4, durum çalışması: Spotify)

Yöntem	Model Üretim süresi (milisaniye)
Genetik	**1014548**
Hungarian	3447
Next Fit	450
Best Fit	414
Worse Fit	415
Sequential Distribution	25
Uniform Distribution	32
Minimum Nodes	20
Manuel Dağıtım	**1980000**

Çizelge 9.7.'te görüldüğü gibi uygulanan algoritmaların çoğu binlerce mikroservisi 1 saniyeye ulaşmadan dağıtmaktadır. Manuel dağıtım yaklaşımı ile bu sürede algoritmik yaklaşımlar kadar verimli dağıtım yapılması zorlu ve uzun bir süreç olmakla birlikte manuel dağıtım süresinin en uzun süre dağıtım yapan genetik algoritmadan daha fazla olduğu görülmektedir.

10. TARTIŞMA

Bu çalışmada mikroservislerin kısıtlı kaynaklara dağıtılmasında verimli alternatiflerin tasarım aşamasında üretilmesi için Model Güdümlü Mühendislik (Model-Driven Engineering-MDE) yaklaşımı önerilmektedir. MDE yaklaşımı ile herhangi bir mikroservis mimarisinin modellenebileceği bir benzetim ortamı hazırlanmaktadır. Bu yaklaşım sayesinde oluşturulan bir sistemin eksiklerini önceden tespit edebilme ve bakımını yapabilme ihtiyacı kolaylıkla giderilebilmektedir. MDE, modelleri ve metamodelleri model dönüşümleri kullanarak üretmekte ve kullanıcıya soyut bir tasarım ortamı sunarak bu amacı gerçekleştirebilmektedir. MDE sayesinde uygulamadaki yapıların basitleştirilmiş soyutlamaları kullanılarak erken tasarım aşamasında çalıştırılabilir bir modele yakın bir yazılım tasarım modeli sağlanmaktadır. MDE'de yer alan görsel yapılar iş ihtiyacını ve teknik sorunların çözümü için tasarımcıya yol göstermektedir. MDE kullanılarak tasarlanan Micro-IDE aracı, mimarinin tasarım aşamasında uygulanabilir dağıtım modelleri üreterek mimariye uygun verimli dağıtım alternatifinin seçimini kullanıcıya bırakmaktadır. Micro-IDE aracının oluşturulması için Eclipse Modelleme Aracı'nda eklenti olarak kullanılabilen EMF [95] ve GMF [96] yazılım araçları kullanılmaktadır. EMF ve GMF'in yanısıra MDE için Acceleo [97], JetBrains MPS [98], Simulink [99] and Sirius [100] gibi bir çok yazılım aracı da alternatif olarak yer almaktadır. Bu modeller arasından Sirius [100] modelleme aracı EMF [95] ve GMF [96] modelleme çerçevelerini kullanarak tasarımcıya karmaşık grafikleri modelleme ortamı sunmaktadır. Fakat bu model zengin bir modelleme deneyimi sağlasa da tasarımcının aracı kullanması için öncelikle değerlendirilecek olan senaryoda gerekli olan tüm bileşenleri manuel olarak tasarlayabilme yeteneğine sahip olması gerekmektedir. Bu durum da tasarımcının Sirius bileşenleri hakkında derin bilgiye sahip olmasını gerektirmektedir. Öte yandan bu çalışma kapsamında EMF, GMF ve Emfatic [92] kullanılarak geliştirilen model güdümlü mimari tasarımcıya basit bir grafik oluşturma çerçevesi sunarken geliştirici tarafında daha fazla karmaşık yapı ve daha fazla kod üretimi gerektirmektedir. Bu bağlamda son kullanıcı açısından değerlendirildiğinde, çalışma kapsamında kullanılan modelleme araçlarının diğer alternatif araçlara göre daha avantajlı olduğu düşünülmektedir. Grafiksel modelleme için Sirius hala umut verici olsa

da karmaşık metamodellerin tanımını sağlamak için Emfatic benzeri otomasyon katmanlarına ihtiyaç duymaktadır.

Verimli dağıtım alternatiflerinin bulunması, sistem içerisinde farklı sorumluluklara sahip binlerce servisten oluşan mikroservis mimarisi için önemli bir süreçtir. Endüstriyel alanda konteyner teknolojisi kullanılarak dağıtım yapılan mikroservis mimarilerinde en verimli dağıtım alternatifini üretebilmek kullanıcının oluşturduğu konfigürasyon dosyasına bağlı olduğundan dolayı kullanıcıdan bağımsız otomatik destekli bir sistem yaklaşımına ihtiyaç duyulmaktadır. Bu problemin üstesinden gelmek için verilen mevcut kaynaklara göre fonksiyonel ve kalite kaygılarını karşılayan, kullanıcıya seçim alternatifi sunan bir otomatik dağıtım aracı önerilerek mikroservis mimarilerinin dağıtımı için sistematik bir yaklaşım sağlanmaktadır. Yaklaşım Eclipse Modelleme Aracı içerisine eklenti olarak kurulan EMF, GMF ve Emfatic çerçevelerini kullanarak Java platform dışında ekstra donanım ve yazılım gerektirmemekte, bu bakımdan diğer otomatik dağıtım araçlarından ayrılmaktadır. EMF kullanılarak birbirlerine bağlı bir şekilde üretilen modeller arasından fiziksel kaynaklara (Mikroservis Altyapı Modeli) ve mikroservislerin çalışma esnasındaki bilgilere (Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli) dayalı olarak uygulanabilir dağıtım alternatifleri algoritmik olarak türetilerek otomatik oluşturulabilmektedir. Çalışmada mikroservislerin kısıtlı kapasiteye sahip fiziksel kaynaklara dağıtılması için kullanılan kısıtlar Kapasite Kısıtlı İş Atama Problemi (Capacitated Task Assignment Problem-CTAP) çözme yaklaşımındaki kısıtlar ile eşleştirilerek problem tanımı mikroservislerin dağıtımı için matematiksel olarak ifade edilmektedir. Tasarım ile ilişkili olarak oluşturulan çözüm algoritmasına ait kısıtlar net bir şekilde belirlendiği için problem bir çok optimizasyon algoritması ile çözülebilir bir şekilde ifade edilmektedir. Önerilen dağıtım yaklaşımı, CTAP çözüm algoritmasına net olarak tanımlanmış girdi kümesi ile bu girdi kümesini kullanarak standart bir çıktı üretilmesini sağlamaktadır.

Araç setinde verimli dağıtım alternatifleri üretmek için farklı algoritmalar uygulanmakta ve bu algoritmalardan farklı performanslar elde edilmektedir. Ayrıca önerilen araç, yeni algoritmaların eklenti olarak kolaylıkla uygulanmasına olanak sağlayarak eklenen algoritma için

gerekli girdi değerleri doğru kullanıldığında, algoritmaya özgü yeni dağıtım alternatifleri sunulabilmektedir. Çalışma kapsamında kullanılan CTAP çözüm yöntemleri dışında geliştirilen sistemin ihtiyaçlarına ve kısıtlarına göre farklı kalite bileşenleri (çalışma zamanı yürütme ortamının dinamik parametreleri, düğümlerin yük dengelemesini, değişen kullanıcı talebini ve ağ trafiğini dikkate alan başka yaklaşımlar) üzerinden eniyileme işlemi yapmak için karınca koloni, sürü optimizasyon algoritması gibi farklı CTAP yöntemlerinin kullanımı desteklenmektedir.

Önerilen yaklaşımı gerçek dünyadan bir durum çalışması ile desteklemek adına Uber'in Taksi Çağırma Sistemi ve Spotify Müzik ve Podcast uygulamasının mikroservis mimarisi ele alınmaktadır. Taksi Çağırma Sistemi için öncelikle 1000 adet servisin 4 adet düğüme aktarıldığı senaryo ele alınarak genetik algoritma tarafından üretilen dağıtım ve manuel dağıtım yaklaşımları karşılaştırılmaktadır. Aynı durum çalışmasında ikinci deney olarak genetik algoritma ile Micro-IDE aracında uygulanan diğer algoritmaların performans karşılaştırılması yapılmaktadır. Spotify durum çalışmasında ise 1361 adet mikroservisin farklı karakteristiklere sahip 4 düğüm üzerinde manuel dağıtım ile tüm algoritmaların ürettikleri modellerin performans karşılaştırılması yapılmaktadır. Çalışmada ayrıca farklı durum çalışmalarında algoritmaların dağıtım modeli üretme zamanları incelenmektedir. Değerlendirme sonucunda, oluşturulan senaryoya, fiziksel kaynak sayısına ve mikroservis sayısına göre algoritmaların üretim zamanlarının farklı olduğu gösterilmektedir.

Araç otomatik dağıtım alternatifi üretmenin yanısıra dağıtım modellerinin manuel olarak tasarlanmasına da olanak tanımaktadır. Bölüm 9.'da manuel dağıtıma göre algoritmik yaklaşımlardan elde edilen iyileşme oranlarına bakıldığında algoritmik yaklaşımların zaman ve performans açısından manuel dağıtıma göre daha iyi performans gösterdiği görülmektedir. Fakat bu durum, manuel dağıtımı yapacak olan uzmana ihtiyaç olmadığı anlamına gelmemektedir. Aslında araç, türetilmiş alternatifleri tasarlayabilen, üretebilen ve değerlendirebilen bir uzman için tamamlayıcı ve destekleyici bir alternatif olmaktadır. Otomatik dağıtım alternatifleri üretildikten sonra gerekirse uzmanın müdahalesi ile optimale daha yakın bir sonuç çıkarmak mümkündür. Yaklaşımın en önemli faydalarından biri sistemin erken analizi ve tasarım aşamasında uygulanabilir dağıtım alternatiflerinin görülebilmesi ve sisteme

en uygun şekilde mimarinin oluşturulmasına olanak tanımasıdır. Dağıtımın geliştirme aşamasında yapılması, herhangi bir değişiklik durumunda proje yaşam döngüsünde yer alan tasarım, implementasyon, dokümantasyon ve test aşamalarına dönülmesine sebep olmakta, bu durum da maliyet ve zaman kayıplarını beraberinde getirmektedir.

Bu çalışmada önerilen yaklaşım ve araç desteği, tasarlanan mikroservis tabanlı bir uygulamanın karakteristiklerine göre verimli bir dağıtım alternatifi bulamadığı durumlarda, yani tasarlanan sistem ve metrikleri belirlenen fiziksel kaynaklara uygun herhangi bir dağıtım alternatifi üretemediği durumlarda tasarımcıya geri bildirimde bulunarak CTAP çözümüne aykırı olan maliyet parametrelerinin iyileştirilmesine olanak tanımaktadır. Tasarımcı bu durumda çalışma zamanı yürütme konfigürasyon metriklerini güncelleyerek aynı senaryo üzerinden tekrar dağıtım yapabilmektedir. Geliştirilen araç bu bağlamda oluşturulan senaryoyu analiz ederek tasarım esnasında dağıtım yapılacak ortamın önceden verimli bir şekilde belirlenmesine yardımcı olmaktadır. Tasarım analizi ve geri bildirim raporu ile birlikte Micro-IDE aracı ile üretilen bir çok dağıtım alternatifinin başarım oranları birbirleri ile karşılaştırılmaktadır.

10.1. Geçerliliğe Yönelik Tehditler

Doktora tezi kapsamında yapılan bu çalışmada Wohlin et al. [101] tarafından geliştirilen standart bir kontrol listesine dayalı dört olası geçerlilik tehdidi (iç geçerlilik-internal validity, yapı geçerliliği-construct validity, sonuç geçerliliği-conclusion validity ve dış geçerlilik-external validity) tartışılmaktadır.

İç geçerlilik: Bu çalışmadaki en kritik geçerlilik tehdidi mikroservislerin dağıtımı için oluşturulan durum çalışmaları hakkında bilgi sahibi olan bir uzmana ulaşılamamasıdır. Bu sebepten dolayı mikroservislerin dağıtımı için sistemi iyi bilen bir uzmanın yapacağı manuel dağıtım, yazar tarafından algoritmik yaklaşımlardan yararlanılarak oluşturulmuştur.

Yapı geçerliliği: Bu geçerleme türü, veri elde etme aşamasında karşılaşılan zorlukların değerlendirilmesi ile ilgilidir [101]. Önerilen yaklaşımı ve aracı test etmek amacı ile belirlenen durum çalışmalarından Taksi Çağırma Sistemi ve Spotify uygulamasında, kullanılan mikroservisler türlerinin tümüne ulaşılamamıştır. Bir diğer husus da sistemlerin çalışması

esnasında dinamik olarak değişen hangi servisten kaç adet bulunduğuna dair örnek bir senaryoya rastlanmamıştır. Bu durumda örnek durum çalışmalarında kullanılan mikroservis adetleri sezgisel olarak yazar tarafından oluşturulmuştur. Bununla birlikte durum çalışmaları için tasarlanan tüm metamodeller, uygulamaların mobil versiyonu üzerinde yer alan bileşenler incelenerek yazar tarafından oluşturulmuştur. Veri elde etme aşamasında karşılaşılan bir diğer zorluk da mikroservis tabanlı uygulamalar geliştiren şirketler ile detaylı bir röportaj yapılamamasıdır. Bu yüzden önerilen bu yaklaşımın ve araç desteğinin şirketler tarafından kullanılabilirliği ve şirketlerin ihtiyacı doğrultusunda farklı parametrelerin kullanılabilirliği irdelenememiştir.

Sonuç geçerliliği: Bölüm 9.'da verilen değerlendirme sonuçları incelendiğinde algoritmaların dağıtım modeli üretim zamanları matematiksel olarak net bir şekilde hesaplanmakta, manuel olarak yapılan dağıtımın model üretim zamanı ise mikroservisleri yerleştiren kullanıcının sistem bilgisine göre değişiklik göstermektedir. Bu bağlamda, çalışma kapsamında geliştirilen algoritmik yaklaşımlar binlerce mikroservisi bir dakikadan kısa sürede dağıtabildiği için bu yaklaşımların manuel dağıtım yaklaşımına göre daha hızlı model ürettiği sonucuna varılmıştır.

Dış geçerlilik: Bu geçerlilik türü, çalışmadan elde edilen sonuçların daha genel bir bağlamda uygulanabilirliği ile ilgilidir. Çalışmada mikroservislerin kısıtlı kapasiteli kaynaklara dağıtımı için belirli parametreler kullanılmıştır. Bu parametreler:

- Mikroservislerin hafıza kapasiteleri
- Düğümlerin hafıza kapasiteleri
- Düğümlerin işlemci gücü
- Mikroservislerin düğümler üzerindeki tahmini çalışma maliyetleri
- Mikroservisler arası iletişim maliyetleri

Bu çalışmada servislerin düğümlere atanması için düğümlerin karakteristik özelliklerinin yer aldığı Mikroservis Altyapı Metamodeli bulut bilişim kaynakları düşünülerek oluşturulduğundan kullanılan ağın bant genişliğinin sınırsız olduğu varsayılmıştır. İlerideki çalışmalarda mikroservis mimarileri üzerine çalışan bir çok firma ile görüşülerek kısıt olarak kullanılacak farklı parametrelerin varlığı araştırılıp algoritmik yaklaşımlara uyarlanması değerlendirilebilir. Yapılan bu çalışmada Kubernetes, Docker Swarm gibi popüler konteyner teknolojilerinde kullanılan parametreler dışında toplam maliyeti önemli ölçüde etkileyerek verimli dağıtımın yapılmasında etken olan mikroservisler arası iletişim maliyetleri kısıt olarak kullanılmıştır.

11. SONUÇ

Mikroservislerin bulut kaynaklarına bağımsız olarak dağıtılması, yüksek talebe sahip servislerin ölçeklendirilerek kaynakların verimli kullanılmasını sağlamaktadır. Bu noktada, mikroservislerin bulut sunuculara bir şekilde dağıtılması önemli bir sorun haline gelmektedir. Mevcut mikroservis dağıtım yaklaşımlarında, dağıtım modelinin oluşturulması uzman kararı ile gerçekleştirilmekte veya genellikle dağıtım aşamasına ertelenmektedir. Uzman değerlendirmesi belirli bir problem boyutuna kadar yeterli olabilse de, uzmanın sistemi çok iyi bilmesi gerekmektedir. Ayrıca manuel dağıtım süreci zaman alıcı olmakla birlikte büyük sistemlerde takip edilemez hale gelmektedir. Bu nedenle, dağıtım modeli tasarımının ve değerlendirmesinin ertelenmesi, geliştirme sürecini olumsuz etkilemekte ve bu durum verimli dağıtım alternatifleri bulmak için proje yaşam döngüsünün tasarım aşamasına geri dönülmesini gerektirmektedir. Verimli dağıtım alternatiflerinin tasarım aşamasında otomatik olarak üretilebilmesi için bu tez çalışmasında, mikroservislerin kaynaklara en iyiye yakın biçimde yerleştirilmesini sağlayacak algoritmik bir yaklaşım sunulmaktadır. Dağıtım problemini çözmek için, tasarımdan parametre çıkarımı yapılmakta, tasarımdan elde edilen parametreler CTAP problemindeki parametreler ile eşleştirilmekte ve CTAP çözücülerinin çıktılarına göre dağıtım modelleri oluşturmak için bir yaklaşım geliştirilmektedir. Bu amaç doğrultusunda Hungarian, Genetik, Minimum Distribution, Sequential Distribution, Uniform Distribution, Next Fit, Best Fit ve Worst Fit olmak üzere bir çok algoritma kullanılmaktadır.

Bu yaklaşımın en önemli faydalarından biri, verimli dağıtım alternatiflerini belirlemek ve tasarım aşamasında sistem mimarisini güncellemek için sistem tasarımının erken analizidir. Uygulama modelini erken tasarım aşaması yerine geliştirme veya sonraki aşamalarda değiştirmek oldukça pahalıdır. Çünkü bu durum, mimari tasarım, ayrıntılı tasarım, uygulama, test öğeleri ve dokümantasyon gibi ürün yaşam döngüsü öğelerinde yeniden çalışmaya yol açacaktır. Bu yeniden yapılan çalışmalar proje maliyetini artıracak ve zamanlamada gecikmelere neden olabilecektir.

Geliştirilen yaklaşım ve araç ailesinin başarımlarının ölçülebilmesi için Eclipse çerçevesine

dayalı bir araç ortamı geliştirilmiştir. Bu araç, bir bulut ortamındaki sanal makineleri, sanal makineler içerisinde çalışacak mikroservisleri ve bunlar arası etkileşimleri tanımlamaya imkan vermektedir. Yaklaşımı doğrulamak adına Uber tarafından geliştirilen bir Taksi Çağırma Sistemi ve Spotify müzik uygulaması, durum çalışması olarak kullanılmıştır. Durum çalışması uygulanırken yaklaşımın ihtiyaç duyduğu mikroservislerin tanımlandığı Mikroservis Tanımlama Modeli, servislerin kullandığı veri yapısını tanımlayan Mikroservis Veri Değişim Modeli, mikroservisler arasındaki iletişim bağlantılarının tanımlandığı Mikroservis İletişim Modeli, mikroservislerin yerleştirildiği Mikroservis Altyapı Modeli ve mikroservis örneklerinin belirlendiği Mikroservis Çalışma Zamanı Yürütme Konfigürasyon Modeli geliştirilmektedir. Bu modellere dayalı olarak CTAP algoritması için gerekli parametreler, verimli yerleştirme alternatifi üretmek için tanımlanmaktadır. Deneysel sonuçlar dağıtım alternatifleri üretim sürelerinin tasarım aşamasında değerlendirme için kabul edilebilir olduğunu göstermektedir. Araç ayrıca, yeni algoritmaların kolaylıkla eklenmesini, oluşturulan dağıtım modellerinin birbirleri ile karşılaştırılmasını, modellerin maliyet ve hafıza kapasiteleri açısından değerlendirilmesini ve analiz edilmesini desteklemektedir.

Bu çalışmanın amacı, algoritmik çözümler kullanarak otomatik dağıtım konfigürasyon alternatiflerini türetmek için sistematik bir yaklaşım sunmaktır. Gelecek çalışmalarda, bu çalışma kapsamında uygulanan algoritmalar dışında, evrimsel hesaplama, doğrusal programlama ve beklenti maksimizasyon algoritmaları gibi CTAP için diğer optimizasyon yaklaşımlarının kullanımı ve performansı araştırılacaktır. Ek olarak, çalışma zamanı yürütme ortamının dinamik parametreleri, düğümlerin yük dengelemesini, değişen kullanıcı talebini ve ağ trafiğini dikkate alan başka yaklaşımlar oluşturmak da hedeflenmektedir. Geliştirilen yaklaşımın deneysel bir bulut ortamında test edilmesi ile gerçek çalışma ortamındaki başarımı ölçülecektir. Bu amaçla, geliştirilen algoritmik yaklaşımların, Amazon ECS, Fargate gibi açık olmayan sistemler yerine, Kubernetes, Docker Swarm, Apache Mesos vb. gibi açık kaynak kodlu sistemlerden en az biri üzerinde gerçeklenerek, sisteme eklenti haline getirilmesi hedeflenmektedir. Böylece çalışan bir bulut ortamında, uygulanan algoritmaların başarımı değerlendirilebileceği gibi literatüre endüstriyel olarak kullanılabilecek bir mikroservis yerleştirme aracı sunulması planlanmaktadır.

KAYNAKLAR

[1] Stephen W Liddle. Model-driven software development. In *Handbook of Conceptual Modeling*, pages 17–54. Springer, **2011**.

[2] Isil Karabey Aksakalli, Turgay Celik, Ahmet Burak Can, and Bedir Tekinerdogan. Systematic approach for generation of feasible deployment alternatives for microservices. *IEEE Access*, 9:29505–29529, **2021**.

[3] T. Mauro. Adopting microservices at netflix: Lessons for architectural design. `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`. (Erişim tarihi: **1 Nisan 2021**).

[4] AWS. Implementing microservices on aws. `https://docs.aws.amazon.com/en_us/whitepapers/latest/microservices-on-aws/introduction.html`. (Erişim tarihi: **1 Nisan 2021**).

[5] Uber. Uber. `https://www.uber.com`. (Erişim tarihi: **31 Temmuz 2019**).

[6] Shop for handmade, vintage, custom and unique gifts. `https://www.etsy.com/`. (Erişim tarihi: **3 Nisan 2021**).

[7] SmartBear. Why you can't talk about microservices without mentioning netflix. `https://smartbear.com/blog/develop/why-you-cant-talk-about-microservices-without-ment/`, **2015**.

[8] A Deb. Application delivery service challenges in microservices-based applications. *Erişim tarihi: Dec*, 10:2020, **2016**.

[9] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings of*

the 9th International Conference on Utility and Cloud Computing, pages 165–174. ACM, **2016**.

[10] Deval Bhamare, Mohammed Samaka, Aiman Erbad, Raj Jain, Lav Gupta, and H Anthony Chan. Multi-objective scheduling of micro-services for optimal service function chains. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, **2017**.

[11] Stefania Costache, Djawida Dib, Nikos Parlavantzas, and Christine Morin. Resource management in cloud platform as a service systems: Analysis and opportunities. *Journal of Systems and Software*, 132:98–118, **2017**.

[12] Ahyoung Kim, Junwoo Lee, and Mucheol Kim. Resource management model based on cloud computing environment. *International Journal of Distributed Sensor Networks*, 12(11):1550147716676554, **2016**.

[13] Swapnil M Parikh, Narendra M Patel, and Harshadkumar B Prajapati. Resource management in cloud computing: classification and taxonomy. *arXiv preprint arXiv:1703.00374*, **2017**.

[14] Puneet Himthani, Amit Saxena, and Manish Manoria. Comparative analysis of vm scheduling algorithms in cloud environment. *International Journal of Computer Applications*, 120(6), **2015**.

[15] Kubernetes. Kubernetes official website. `https://kubernetes.io/`. (Erişim tarihi: **31 Temmuz 2019**).

[16] Docker Swarm. Swarm mode overview. `https://docs.docker.com/engine/swarm/`. (Erişim tarihi: **29 Mart 2021**).

[17] Apache Mesos. Program against your datacenter like it's a single pool of resources. `http://mesos.apache.org/`. (Erişim tarihi: **29 Mart 2021**).

[18] D.L.Ek Martin. Spotify. `https://www.spotify.com/`. (Erişim tarihi: **17 Şubat 2020**).

[19] Taner Pirim. A hybrid metaheuristic algorithm for solving capacitated task allocation problems as modified xqx problems. **2006**.

[20] Miika Kalske. Transforming monolithic architecture towards microservice architecture. **2019**.

[21] Anastasia D. Best architecture for an mvp: Monolith, soa, microservices, or serverless? `https://rubygarage.org/blog/monolith-soa-microservices-serverless`. (Erişim tarihi: **15 Kasım 2019**).

[22] Kasun Indrasiri and Prabath Siriwardena. *Microservices for the Enterprise: Designin, Developing and Deploying*. Apress 1st edition, **2018**.

[23] Mark Richards. Microservices vs. service-oriented architecture. `https://www.oreilly.com/radar/microservices-vs-service-oriented-architecture/`. (Erişim tarihi: **15 Kasım 2019**).

[24] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. IEEE, **2015**.

[25] IBM. Soa vs. microservices: What's the difference? `https://www.ibm.com/cloud/blog/soa-vs-microservices#`. [Online; Erişim tarihi 03-04-2021].

[26] C. Richordson. Microservices from design to deployment. `https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/`. (Erişim tarihi: **31 Mart 2021**).

[27] AWS. Amazon machine images (ami). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html. (Erişim tarihi: **31 Mart 2021**).

[28] AWS. Aws lambda. https://aws.amazon.com/tr/lambda/. (Erişim tarihi: **31 Mart 2021**).

[29] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. Optimal and automated deployment for microservices. In *International Conference on Fundamental Approaches to Software Engineering*, pages 351–368. Springer, **2019**.

[30] Apache. Apache mesos. http://mesos.apache.org/. (Erişim tarihi: **31 Temmuz 2019**).

[31] Adalberto R Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S Rosa. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, 10(1):4, **2019**.

[32] Kapil Bakshi. Microservices-based software architecture and approaches. In *2017 IEEE Aerospace Conference*, pages 1–8. IEEE, **2017**.

[33] Yipei Niu, Fangming Liu, and Zongpeng Li. Load balancing across microservices. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 198–206. IEEE, **2018**.

[34] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. Performance analysis of restful api and rabbitmq for microservice web application. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 257–259. IEEE, **2018**.

[35] Muhammad Alam, Joao Rufino, Joaquim Ferreira, Syed Hassan Ahmed, Nadir Shah, and Yuanfang Chen. Orchestration of microservices for iot using docker and edge computing. *IEEE Communications Magazine*, 56(9):118–123, **2018**.

[36] Antonin Smid, Ruolin Wang, and Tomas Cerny. Case study on data communication in microservice architecture. In *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, pages 261–267. **2019**.

[37] RabbitMQ. Rabbitmq. `https://www.rabbitmq.com/`. (Erişim tarihi: **1 Nisan 2021**).

[38] Apache. Kafka. `https://kafka.apache.org`. (Erişim tarihi: **1 Nisan 2021**).

[39] Activemq. Activemq. `http://activemq.apache.org/`. (Erişim tarihi: **1 Nisan 2021**).

[40] Michael Hofmann, Erin Schnabel, Katherine Stanley, et al. *Microservices Best Practices for Java*. IBM Redbooks, **2017**.

[41] Deval Bhamare, Mohammed Samaka, Aiman Erbad, Raj Jain, and Lav Gupta. Exploring microservices for enhancing internet qos. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3445, **2018**.

[42] Eberhard Wolff. *Microservices: flexible software architecture*. Addison-Wesley Professional, **2016**.

[43] Microservice issues, challenges, and hurdles. `https://dzone.com/articles/microservice-issueschallenges-and-hurdles`, **2018**. (Erişim tarihi: **17 Haziran 2021**).

[44] Communication between microservices: How to avoid common problems. `https://stackify.com/communication-microservices-avoid-common-problems/`, **2017**. (Erişim tarihi: **17 Haziran 2021**).

[45] Challenges of micro-service deployments. `http://techtraits.com/microservice.html`, **2016**. (Erişim tarihi: **18 Haziran 2021**).

[46] Benjamin Benni, Sébastien Mosser, Philippe Collet, and Michel Riveill. Supporting micro-services deployment in a safer way: a static analysis and automated rewriting approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1706–1715. **2018**.

[47] Vindeep Singh and Sateesh K Peddoju. Container-based microservice architecture for cloud applications. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 847–852. IEEE, **2017**.

[48] Communicating with microservices. `https://medium.com/high-alpha/communicatingwith-microservices-92a2c59d697c`, **2017**. (Erişim tarihi: **15 Haziran 2021**).

[49] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, **2014**.

[50] Dong Guo, Wei Wang, Guosun Zeng, and Zerong Wei. Microservices architecture based cloudware deployment platform for service computing. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 358–363. IEEE, **2016**.

[51] Päivi Parviainen, Juha Takalo, Susanna Teppola, and Maarit Tihinen. Model-driven development: Processes and practices. **2009**.

[52] Anthony MacDonald, Danny Russell, and Brenton Atchison. Model-driven development within a legacy system: an industry experience report. In *2005 Australian Software Engineering Conference*, pages 14–22. IEEE, **2005**.

[53] Stephen J Mellor, Tony Clark, and Takao Futagami. Model-driven development: guest editors' introduction. ieee software, 20 (5). pp. 14-18. issn 0740-7459. *IEEE software*, 20(5):14–18, **2003**.

[54] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE, **2007**.

[55] OMG. Omg: Data distribution service (dds). https://en.wikipedia.org/wiki/Data_Distribution_Service. (Erişim tarihi: **31 Mart 2021**).

[56] Scott W Ambler. Examining the model driven architecture (mda), **2010**.

[57] J. Miller and J. Mukerji (Eds.). Model driven architecture (mda). http://www.omg.org/cgi-bin/doc?ormsc/01-07-01, **2001**. (Erişim tarihi: **26 Mart 2021**).

[58] OMG. Mda guide version 1.0.1. https://www.omg.org/news/meetings/workshops/UML_2003_Manual/00-2_MDA_Guide_v1.0.1.pdf, **2003**. (Erişim tarihi: **26 Mart 2021**).

[59] JF Overbeek. *Meta Object Facility (MOF): investigation of the state of the art*. Master's thesis, University of Twente, **2006**.

[60] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, **2009**.

[61] Zsolt Lattmann, James Klingler, Patrik Meijer, Ted Bapty, Sandeep Neema, and Jason Scott. Institute for software-integrated systems. *ISIS*, 15:106, **2015**.

[62] rgronback. Mindmap.ecore model. https://github.com/eclipse/gmf-tooling/blob/master/examples/org.eclipse.gmf.examples.mindmap.lite/model/mindmap.genmodel, **2007**. [Online; Erişim tarihi 10-May-2019].

[63] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Microart: A software architecture recovery tool for maintaining microservice-based systems. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 298–302. IEEE, **2017**.

[64] Thomas F Düllmann and André van Hoorn. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 171–172. **2017**.

[65] Turgay Çelik, Bedir Tekinerdogan, and Kayhan M İmre. Deriving feasible deployment alternatives for parallel and distributed simulation systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 23(3):18, **2013**.

[66] Turgay Celik and Bedir Tekinerdogan. S-ide: A tool framework for optimizing deployment architecture of high level architecture based simulation systems. *Journal of Systems and Software*, 86(10):2520–2541, **2013**.

[67] Bedir Tekinerdogan, Turgay Çelik, and Ömer Köksal. Generation of feasible deployment configuration alternatives for data distribution service based systems. *Computer Standards & Interfaces*, 58:126–145, **2018**.

[68] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods*, pages 194–210. Springer, **2016**.

[69] Augusto Ciuffoletti. Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163–172, **2015**.

[70] Christian Berger, Björnborg Nguyen, and Ola Benderius. Containerized development and microservices for self-driving vehicles: Experiences & best practices. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 7–12. IEEE, **2017**.

[71] Jean-Philippe Gouigoux and Dalila Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 62–65. IEEE, **2017**.

[72] Tianlei Zheng, Yuqun Zhang, Xi Zheng, Min Fu, and Xiao Liu. Bigvm: A multi-layer-microservice-based platform for deploying saas. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 45–50. IEEE, **2017**.

[73] Davide Profeta, Nicola Masi, Domenico Messina, Davide Dalle Carbonare, Susanna Bonura, and Vito Morreale. A novel micro-service based platform for composition, deployment and execution of bda applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 182–185. IEEE, **2019**.

[74] Pablo Chico de Guzmán, Felipe Gorostiaga, and César Sánchez. Pipekit: A deployment tool with advanced scheduling and inter-service communication for multi-tier applications. In *2018 IEEE International Conference on Web Services (ICWS)*, pages 379–382. IEEE, **2018**.

[75] Xili Wan, Xinjie Guan, Tianjing Wang, Guangwei Bai, and Baek-Yong Choi. Application deployment using microservice and docker containers: Framework and optimization. *Journal of Network and Computer Applications*, 119:97–109, **2018**.

[76] Juliana Carvalho, Dario Vieira, and Fernando Trinta. Greedy multi-cloud selection approach to deploy an application based on microservices. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 93–100. IEEE, **2019**.

[77] Autonomic Computing et al. An architectural blueprint for autonomic computing. *IBM White Paper*, 31(2006):1–6, **2006**.

[78] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. In *International Symposium on Formal Methods for Components and Objects*, pages 142–164. Springer, **2010**.

[79] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, **2009**.

[80] C Robson. Real world research blackwell. *2º edição*, **2002**.

[81] Matthijs Neppelenbroek, Matthias Lossek, Rik Janssen, and Tim de Boer. Twitter an architectural review. `http://www.timdeboer.eu/paper_publishing/Twitter_An_Architectural_Review.pdf`, **2011**.

[82] Chris Richardson. Building microservices: Inter-process communication in a microservices architecture. `https://www.nginx.com/blog/building-microservices-inter-process-communication`. (Erişim tarihi: **1 Nisan 2021**).

[83] K. Goldsmith. Microservices at spotify. `https://gotocon.com/dl/goto-berlin-2015/slides/KevinGoldsmith_MicroservicesSpotify.pdf`. (Erişim tarihi: **17 Şubat 2020**).

[84] E. Wilde and C. Pautasso. Aws lambda. `https://restfulapi.net/`. (Erişim tarihi: **31 Mart 2021**).

[85] gRPC. A high performance, open-source universal rpc framework. `https://grpc.io/`. (Erişim tarihi: **12 Eylül 2019**).

[86] Amazon. Amazon simple notification service. `https://aws.amazon.com/sns/`. (Erişim tarihi: **01 Nisan 2021**).

[87] Amazon. Amazon simple queue service. `https://aws.amazon.com/tr/sqs/`. (Erişim tarihi: **1 Nisan 2021**).

[88] Java. Oracle. `https://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html`. (Erişim tarihi: **1 Nisan 2021**).

[89] Shahir Daya, Nguyen Van Duy, Kameswara Eati, Carlos M Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, Valerie Lampkin, Marcelo Martins, et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, **2016**.

[90] GraphQL. A query language for your api. `https://graphql.org/`. (Erişim tarihi: **12 Eylül 2019**).

[91] AWS. Pub/sub messaging. `https://aws.amazon.com/tr/pub-sub-messaging/`. (Erişim tarihi: **12 Eylül 2019**).

[92] IBM alphaWorks. Emfatic language for emf development. `http://www.alphaworks.ibm.com/tech/emfatic`. (Erişim tarihi: **6 Nisan 2020**).

[93] Dimitrios S Kolovos, Antonio García-Domínguez, Louis M Rose, and Richard F Paige. Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software & Systems Modeling*, 16(1):229–255, **2017**.

[94] Abbas Mehrabi, Saeed Mehrabi, and Ali D Mehrabi. An adaptive genetic algorithm for multiprocessor task assignment problem with limited memory. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 2, page 115. **2009**.

[95] Eclipse modeling framework (emf). `https://www.eclipse.org/modeling/emf/`, **2021**. (Erişim tarihi: **17 Mayıs 2021**).

[96] Gmf tooling. `https://www.eclipse.org/gmf-tooling/`, **2021**. (Erişim tarihi: **17 Mayıs 2021**).

[97] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud, and Freddy Allilaire. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, 2:157, **2006**.

[98] Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 165–168. **2013**.

[99] Steven T Karris. *Introduction to Simulink with engineering applications*. Orchard Publications, **2006**.

[100] Eclipse sirius. `https://www.eclipse.org/sirius/`, **2021**. (Erişim tarihi: **17 Mayıs 2021**).

[101] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10. **2014**.

EKLER

1. Verimli mikroservis dağıtım modeli üretmek için geliştirilen algoritma

```
1: GENERATE_EFFICIENT_DEPLOYMENT (phy_resources, exec_config)
2:     processors← EXTRACT_PROCESSORS (phy_resources)
3:     tasks← EXTRACT_TASKS (exec_config)
4:     allocation_table←EXECUTE_CTAP (tasks, processors)
5:     CREATE_DEPLOYMENT_MODEL (allocation_table)
6:
7: EXTRACT_PROCESSORS (resources)
8:     create empty list processors
9:     for each node in resources do
10:        processors← CREATE_PROCESSOR (node)
11:        append processor to processors
12:    end for
13:    return processors
14:
15: EXTRACT_TASKS (exec_config)
16:    create empty list tasks
17:    for each source_module_inst in exec_config do
18:        tasks← CREATE_TASK (source_module_inst)
```

```
19:             for each target_module_inst in exec_config do
20:                 comm_cost← GET_COMM_COST(source_module_inst,target_module_inst)
21:                 SET_COMM_COST(task, comm_cost,target_module_inst.ID)
22:             end for
23:             append task to tasks
24:         end for
25:         return tasks
26:
27:  CREATE_DEPLOYMENT_MODEL (allocation_table)
28:         for each task in allocation_table do
29:             deployed_module←CREATE_DEPLOYED_MODULE(task)
30:             deployed_module.processor←GET_PROCESSOR(allocation_table,task)
31:         end for
32:
33:  GET_COMM_COST (source_module_inst, target_module_inst)
34:         communication_cost=0
35:         if source_module_inst not equals target_module_inst then
36:         for each publication of source_module_inst do
37:             data_exchange_object←publication.data_exchange_object
38:             if IS_SUBSCRIBER (target_module_inst, data_exchange_object)
39:             then
40:                 communication_cost += CALCULATE_COST (publication)
41:             end if
42:         end for
43:        end if
44:        return communication_cost
45:
46:  IS_SUBSCRIBER (target_module_inst, data_exchange_object)
47:         module←target_module_inst.related_module
```

```
48:     if module subscribes to data_exchange_object or a parent of it
49:     then return TRUE
50:     end if
51:     else return FALSE
52:     end else
53:
54: CALCULATE_COST (publication)
55:     data_exchange_object←publication.data_exchange_object
56:     update_rate←publication.update_rate
57:     return update_rate x SIZEOF (data_exchange_object)
58:
59: SIZEOF (data_exchange_object)
60:     size=0
61:     if data_exchange_object has a parent_object then
62:         size += SIZEOF (parent_object)
63:     end if
64:     if data_exchange_object is an Object Class then
65:         for each attribute of data_exchange_object do
66:             size += SIZEOF (attribute.datatype)
67:         end for
68:     end if
69:     return size
70:
71: SIZEOF_DATATYPE (parameter.datatype)
72:     size=0
73:     if parameter.datatype is a Message then
74:         size= ((Message) parameter.datatype).size
75:     end if
76:     else then
77: // calculates the size of enumerated, array, fixed, list and
```

```
78:  // variant data types recursively until reaching message type elements
79:          end else
80:          return size
```