T.C. SANAYİ VE TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

MONOLİTİK MİMARİDEN MİKROSERVİS MİMARİYE DÖNÜŞÜM

ARAŞTIRMA SERİSİ - SAYI 11

BİLGEM

YAZILIM TEKNOLOJİLERİ ARAŞTIRMA ENSTİTÜSÜ

Simge ve Kısaltmalar

Kısaltmalar	Açıklama
TÜBİTAK	Türkiye Bilimsel ve Teknolojik Araştırma Kurumu
BİLGEM	Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezi
YTE	Yazılım Teknolojileri Araştırma Enstitüsü
CI/CD	Continuous Integration/Continuous Delivery (Sürekli Entegrasyon / Sürekli Dağıtım)
API	Application Programming Interface (Uygulama Programlama Arayüzü)
ACID	Atomicity (Bölünemezlik), Consistency (Tutarlılık), Isolation (İzolasyon), Durability (Dayanıklılık)
DevOps	Development Operations (Geliştirme Operasyonları)

Yazar
Sabiha Deniz ACUN

Yayın Koordinatörü
Elif ŞENYİĞİT

Editörler
İhsan Baran SÖNMEZ
Sevinç KARAKAŞ
Tuğçe YILMAZ

Tasarım
Şeyma KOÇER

İletişim: 0(312) 289 92 22 - yte.bilgi@tubitak.gov.tr

https://bilgem.tubitak.gov.tr/yte/

İçindekiler

Önsöz

TÜBİTAK BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE), 2012 yılından bu yana yazılım teknolojilerinde Ar-Ge faaliyetleri yürüten bir araştırma kuruluşudur. Araştırma faaliyetlerinde elde ettiği birikimini stratejik, hassas ve kritik projeler yürüterek kamu adına hayata geçirmekte; kurumlarımıza dijital dönüşüm, yazılım geliştirme teknolojileri ve kalite süreçleri konusunda danışmanlık vermektedir.

TÜBİTAK BİLGEM YTE tarafından hazırlanan Araştırma Serisi ile kurum içi içerik üretme çalışmalarının yaygınlaştırılması ve hazırlanan içeriklerin sektörün erişimine açılması amaçlanmaktadır. Araştırma Serisi'nde yayınlanan çalışmalar TÜBİTAK BİLGEM YTE çalışanlarının projelerde elde ettiği bilgi birikimini paylaşmak adına derlenmiştir. Bu çalışmalar ile ülkemizin yazılım sektörüne katkı sağlanılması hedeflenmektedir.

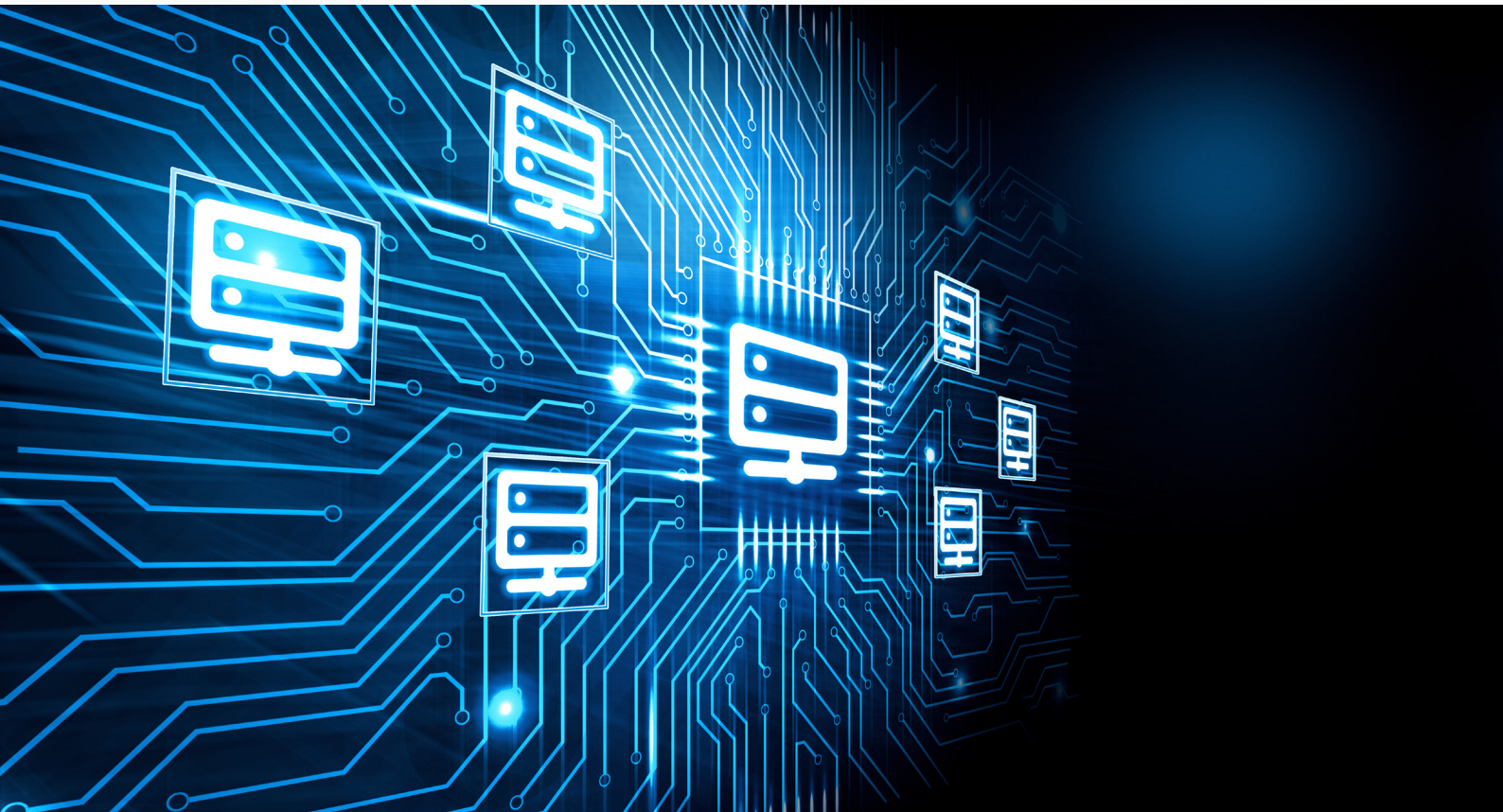
Giriş

Günümüzün hızla değişen teknoloji pazarında, işletmeler yazılım geliştirme ve iş süreçlerini optimize etmenin yollarını aramaktadır. Bu bağlamda, monolitik mimariden mikroservis mimarisine geçiş, organizasyonlar için önemli bir dönüşümü temsil etmektedir. Monolitik mimari, uzun yıllardır yazılım geliştirmenin belirleyici bir modeli olarak hizmet vermiştir ve birçok projede başarılı bir şekilde kullanılmıştır. Ancak teknolojinin hızla evrildiği bir çağda, organizasyonlar daha çevik, ölçeklenebilir ve hızlı bir şekilde geliştirme yapma ihtiyacı duymaktadır.

Monolitik mimarinin, yazılım sistemlerini tek bir büyük ve bütünsel bir yapıda toplaması, başlangıçta avantajlı gibi görünse de zamanla bazı kısıtlamaları ortaya çıkarır. Bu kısıtlamalar, büyük ölçekli projelerde karmaşıklığı artırabilir, yeni özelliklerin eklenmesini zorlaştırabilir ve ölçeklenebilirlik sorunlarına yol açabilir. Ayrıca, monolitik mimari, uygulamanın belirli bir teknoloji ekosistemine bağımlı hale gelmesine neden olabilir, bu da teknolojik yeniliklere adapte olmayı güçleştirebilir.

Mikroservis mimarisi, bu tür zorluklara karşı bir alternatif sunar. İşlevselliğe göre bağımsız geliştirilen ve çalışan hizmetleri içeren bu yaklaşımın, çeviklik, sürekli dağıtım ve daha iyi ölçeklenebilirlik gibi avantajları vardır. Mikroservis mimarinin avantajları göz önünde bulundurulduğunda organizasyonlar monolitik mimariden mikroservis mimariye geçiş yapmak isteyebilirler. Ancak, monolitik mimariden mikroservis mimarisine geçiş büyük bir adımı temsil eder. Bu geçiş, teknik ve organizasyonel zorluklarla doludur ve dikkatli bir planlama ve uygulama gerektirir.

Bu çalışma kapsamında, monolitik mimariden mikroservis mimariye dönüşümde karşılaşılan zorluklara ve bu zorlukların çözümlerine yer verilmiştir.



Monolitik Mimari

Monolitik mimari, bir yazılım uygulamasının tüm bileşenlerinin tek bir büyük ve bütünsel bir yapı içinde entegre edildiği geleneksel bir yazılım geliştirme yaklaşımını ifade eder. Bu yaklaşım, uygulama içindeki tüm işlevselliği tek bir monolitik kod tabanı içinde barındırır. Genellikle aynı teknoloji yığınına dayanır ve tüm bileşenler birbirine sıkıca bağlıdır.

Monolitik mimaride, uygulamanın tüm parçaları tek bir paket halinde dağıtılır, bu da uygulamanın dağıtımını ve bakımını kolaylaştırabilir. Ayrıca, tüm bileşenler aynı veri tabanını paylaştığından veri bütünlüğü kolayca sağlanabilir.

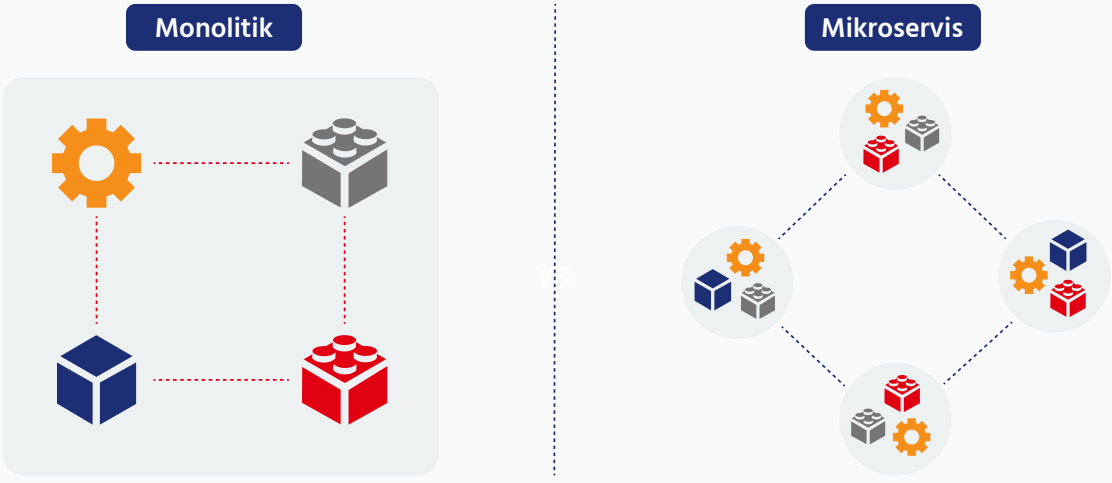
Mikroservis Mimari

2005 yılında ilk kez Dr. Peter Rogers tarafından Bulut Bilişim konferansında "micro web services" olarak telaffuz edilmesi ile hayatımıza girmiştir. "Microservices" terimi ise Mayıs 2011 yılında Viyana yakınlarında gerçekleşen yazılım mimarileri seminerinde kaydedilmiştir ve günümüzde bu ad ile kullanılmaya devam etmektedir.

Mikroservis mimarisi, basitçe mikroservisler olarak bilinen bağımsızca dağıtımı (deploy) yapılabilen servisler dizisine dayanan bir mimari yöntemdir. Bu servisler, kendi iş mantığı ve belirli bir hedefi olan kendi veri tabanlarına sahiptir. Güncelleme, test, dağıtım ve ölçeklendirme, her servis içinde gerçekleşir.

Mikroservisler karmaşıklığı azaltmaz, ancak karmaşıklığı görünür kılar ve görevleri birbirinden bağımsız olarak çalışan ve genel bütüne katkıda bulunan daha küçük süreçlere ayırarak daha yönetilebilir hale getirir.

Mikroservisleri benimsemek genellikle DevOps ile el ele gider, çünkü bunlar kullanıcı gereksinimlerine hızlı bir şekilde uyum sağlama olanağı sunan sürekli teslimat (continuous delivery) uygulamalarının temelini oluştururlar.



Monolitik Mimari ve Mikroservis Mimari Karşılaştırması

Monolitik Mimariden Mikroservis Mimariye Dönüşümün Nedenleri

Kuruluşlar, genel itibariyle teknolojiye dayalı sağlam ve güçlü çözümlere ihtiyaç duyar. Bu nedenle, yazılım geliştiriciler, yazılım ürünlerinin kaynak açısından verimli olmasına ve işlevsel gereksinimlerden oluşmasına yardımcı olan farklı türlerde mimariler tasarlamış ve uygulamıştır. Bazı mimariler, modüllerini tek katmanda veya farklı katmanlarda dağıtılmış olarak tutar. Yazılım üretiminde geleneksel olarak sanal makinelerle birlikte kullanılan monolitik mimari, yazılım sistemlerinin ortaya çıkışından bu yana küçük ve büyük ölçekli projeler için başarılı ve etkili bir formül olmuştur (Tapia, Mora, Fuertes, Aules, Flores and Theofilos, 2020).

Fakat teknolojinin hızla değişmesi yeni mimari modellerinin ortaya çıkmasına ve bu modelleri uygulamaya teşvik etmesine neden olmuştur. Bu mimari modellerden biri de günümüzde en çok tercih edilen ve potansiyel birçok probleme çözüm bulan mikroservis mimarisidir.

Organizasyonlar üretmiş oldukları yazılımlarda kullandıkları monolitik mimari tasarımından dolayı sık sık zorluklarla karşılaşmaktadır. Bu zorluklar aşağıdaki gibi olabilir:

- Zamanla sistemde karmaşıklığın artması ve üretkenliğin düşmesi
- Yeni özellikler (feature) ekleme ve bunların optimizasyonlarının zorlaşması
- Sınırlı ölçeklenebilirlik
- Yapılandırma (build) sürelerinin ve test sürelerinin uzaması
- Uygulamanın test edilebilirliğinin azalması
- Projenin tek veri tabanı kullanması sonucu çıkan optimizasyon ve performans sıkıntıları
- Uygulamaların belirli bir dil/teknoloji ekosistemine bağımlı hale gelmesi
- Dağıtım/teslimat (CI/CD) süreçlerinin uygulanmasının zorlaşması

Bu ve benzeri bazı zorluklar, monolitik mimarinin uygulandığı sistemler için mikroservis mimari tasarımı çözüm olarak sunulabilmektedir. Fakat monolitik mimarinin, mikroservis mimariye dönüşümü için güçlü sebepler olması gerekmektedir. Bu sebepleri aşağıdaki gibi özetleyebiliriz:

- Kurumların çevik olma gibi hedefleri varsa ve bu sebeple kolay, hızlı ve devamlı ufak sürümler (release) çıkarılması isteniyorsa,
- Farklı makinelerde çalışan, farklı teknoloji/dil kullanılarak kendi işlevselliğine göre bağımsız geliştirilebilen ve tek bir göreve odaklanan alt etki alanları (subdomain) olması isteniyorsa,
- Dağıtım/teslimat süreçlerinin hızlı bir şekilde ele alınabilmesi isteniyorsa,
- Anlaşılması kolay kod yapısı ile projeye yeni katılacak geliştiricilerin kolay adapte olması isteniyorsa.

Mikroservis Mimariye Dönüşümde Karşılaşılan Zorluklar

Kurumlar bünyelerinde yer alan yazılımlarda gerek karşılaştıkları zorluklar gerekse güncel teknolojilere ayak uydurma isteğinden dolayı monolitik mimariden mikroservis mimariye dönüşüm çalışmaları yapmaktadır. Fakat bu dönüşüm projeler için kolay olmamakla birlikte organizasyonların zorluklarla karşılaşmasına sebep olmaktadır. Bu zorluklar teknik zorluklar ve organizasyonel zorluklar olarak ikiye ayrılır. Her iki zorluk türü arasında farklılıklar olsa da eşit derecede önemlidir. Uygulama geliştirmeye sıfırdan başlanması ile mevcut büyük bir kod tabanının monolitik mimariden mikroservis mimariye dönüştürülmesi arasındaki zorluklar biraz daha farklıdır. Bu bölümde, mevcut monolitik uygulamanın mikroservis mimarisine doğru yeniden düzenlenmesi ile ilgili zorluklara odaklanılmıştır. Aynı zamanda bu zorlukların çoğu, mikroservisler ile yeni bir proje geliştirirken de ele alınması gereken durumlardır.

1. Teknik Zorluklar

Yazılım geliştiricilerin monolitik mimariden mikroservis mimariye geçerken karşılaştıkları teknik sorunlar aşağıdaki gibi özetlenebilir:

- Monolitik mimaride yer alan servislerin ayrılması,
- Monolitik mimarideki veri tabanı yapısının ayrıştırılması,
- İşlem (transaction) bütünlüğü yönetiminin ele alınması.

1.1. Monolitik Mimaride Yer Alan Servislerin Ayrılması

En büyük problemlerden biri monolitik mimarideki servislerin ayrılmasıdır. Bu oldukça fazla zaman ve efor alabilir (Kalske, M¨akitalo, and Mikkonen, February 2018). Var olan monolitik bir sistemi parçalamak söz konusu olduğunda, mantıksal parçalamaya ihtiyaç bulunmaktadır. Bu noktada, yazılım geliştirme sürecinde karmaşık iş alanlarını (business domain) ve gereksinimleri daha iyi anlamak ve bu anlayışı yazılım tasarımına yansıtmak için kullanılan alan odaklı tasarım (domain-driven design) metodolojisini kullanmak işe yaramaktadır.

Bir yazılımdaki iş süreçlerinin, kavramların ve gereksinimlerin doğru şekilde yansıtıldığı etki alanı (domain) modelini geliştirmek, etki alanını ayırma konusunda öncelikleri belirlemede yardımcı olur. Alberto Brandolini tarafından oluşturulan “olay fırtınası (event storming)” egzersizi kullanılarak, teknik ve teknik olmayan paydaşlar bir araya gelerek ortak bir etki alanı tanımlanır. Bu egzersizin çıktısı sadece model değil, aynı zamanda modelin paylaşılan anlayışıdır.

Yazılım sistemi içerisinde belirli bir alanın veya bağlamın sınırlarını tanımlayan "sınırlı bağlam alanı (bounded context)" mikroservis sınırlarını tanımlamak için başlangıç noktalarını yaratır. Her bir sınırlı bağlam alanı (bounded context), ayırma işlemindeki potansiyel en küçük birimi temsil eder. Bu sayede bu birimleri baz alarak önceliklendirme yapılabilir.

Bağımlılıkları olmayan ve daha kolay anlaşılır sistem davranışına sahip olan sınıflar; mevcut monolitik sistemin üzerinde yapılması gereken değişiklikleri azaltır. Bu sebepten ideal bir başlangıç noktası olarak, monolitik uygulamada ayrılacak ilk servis adaylarından biri olarak gevşek bağlı (loosely coupled) bileşenler seçilebilirler. Ancak, diğer servis bileşenlerinin özelliklerinin de değerlendirilmesi gerekir. Bu değerlendirmeler şunlara göre yapılabilir:

- Bileşenin teknik borcu, taşınan mikroservis ile birlikte bu borçları almaktan kaçınmak için daha düşük olmalıdır.
- Bileşenin test kapsamı, geçiş sonrasında bileşenlerin test edilmesinin kolay olması için yüksek olmalıdır.
- Bileşenler, iş birimi için yüksek değerde olmalıdır. Bu sayede daha hızlı dağıtım döngüleri ve yerine getirilen ölçeklenebilirlik gereksinimleri gibi geçişin faydaları belirgin şekilde görülür.

Ayrılacak aday servis belirlenip bunu mikroservis hizmeti olarak geliştirmeye başlamak için kullanılabilecek desenler (pattern) mevcuttur. Bunlar:

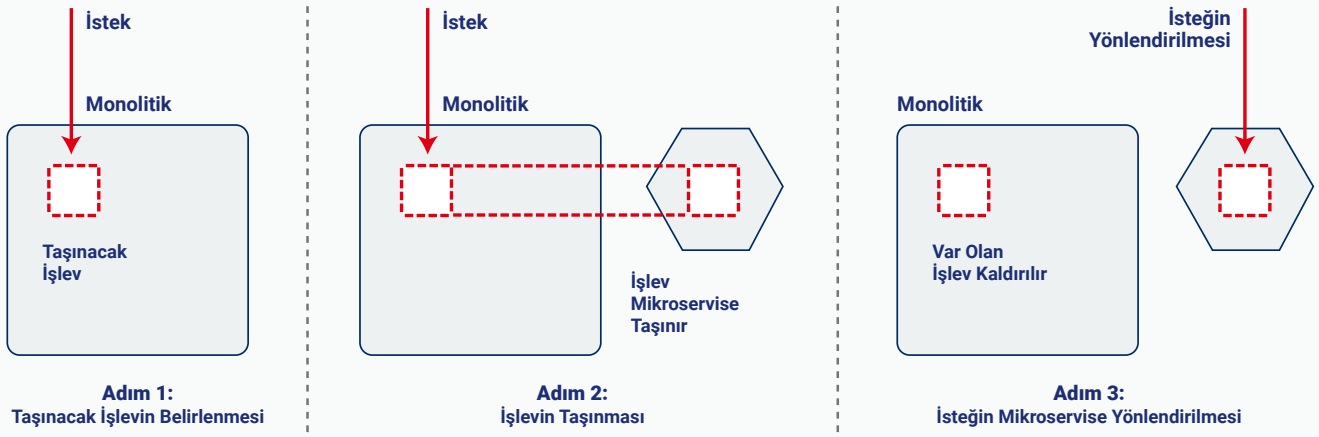
- Strangler Fig Application
- Branch by Abstraction
- Parallel Run
- Decorating Collaborator
- Change Data Capture

1.1.1. Strangler Fig Application

Monolitik mimariden mikroservis mimariye dönüşümde adı en çok bilinen ve en çok kullanılan desendir. Monolitik mimariden her seferinde seçilen bir işlevi kademeli bir şekilde mikroservis mimarisine dönüştürerek, zaman içinde tüm işlevleri monolitik mimariden mikroservis mimarisine taşır.

Bu desenin temel faydası, kısa bir süre içinde yeni bir sisteme aşamalı geçiş yapma hedefini desteklemesidir. Ayrıca; bu desen, dönüşümü duraklatma ve hatta tamamen durdurma yeteneği sunarken, şu ana kadar sunulan yeni sistemden hala faydalanılmasına olanak tanır.

Strangler fig deseninin uygulanması üç adıma dayanır. İlk olarak, dönüştürülmek istenilen mevcut sistemin parçaları tanımlanır. Daha sonra bu işlev yeni mikroserviste uygulanır. En sonunda da yeni mikroservis hazır olduğunda, çağrılar monolitik sistemden yeni mikroservise yönlendirilir.



Şekil 2. Strangler Fig Pattern Örneği

Strangler fig deseni uygulandığında, sadece yeni uygulama mimarisi için aşamalı adımlar atmaya çalışmakla kalınmaz, aynı zamanda her adımın kolayca geri alınabilir olduğundan emin olunur ve her aşamanın riski azaltılır.

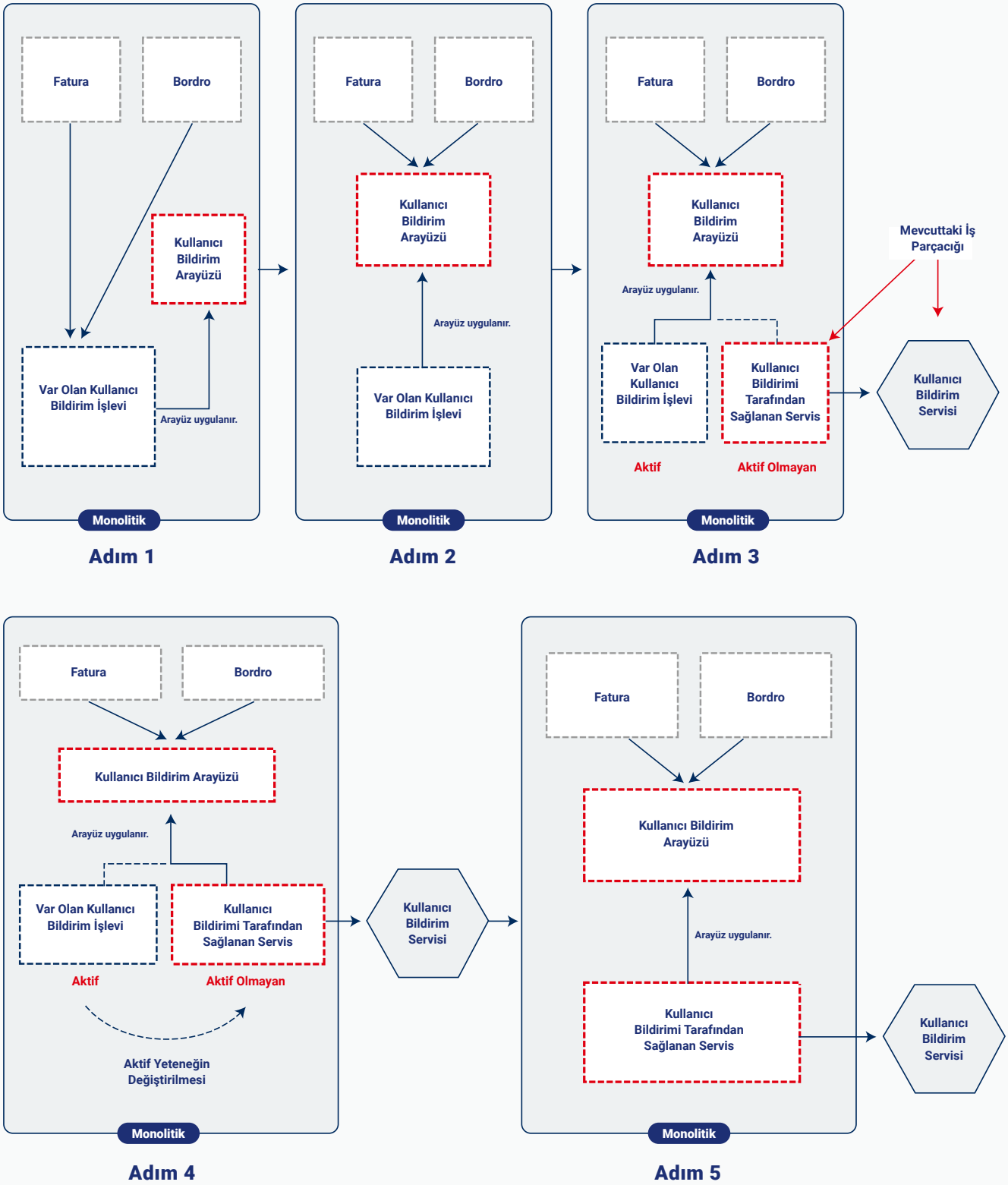
1.1.2. Branch by Abstraction

Uygun görülen işlevi, mikroservis olarak çıkarmayı gerçekleştirmek için, mevcut sistemde değişiklikler yapılması gerekir. Bu değişiklikler uygulama için çok önemli olabilir ve aynı anda kod yapısı üzerinde çalışan diğer geliştiricilere müdahale edilmesine sebep olabilir. Bir yandan değişiklikler aşamalı adımlarla yapılmak istenirken öte yandan, kod yapısının diğer alanlarında çalışan geliştiricilere müdahalenin oldukça az olması istenir. Doğal olarak, bu durum değişikliğin hızlı bir şekilde tamamlanmak istenmesine yol açar. Bu durumda uygulanabilecek en uygun desen Branch by Abstraction desenidir.

Branch by Abstraction deseni kaynak kodun bulunduğu dala (branch) başvurmadan, kod yapısının belirli bir parçasında aşamalı değişiklik yapılmasını sağlayarak, kodun diğer parçalarında çalışan geliştiricilere müdahaleyi en aza indir. Bu desen aynı kod sürümünde çok fazla kesintiye neden olmadan uygulamaların birbirleriyle güvenli bir şekilde yan yana var olabilmeleri için mevcut kod yapısı üzerinde değişiklikler yapmayı hedefler.

Branch by Abstraction deseninin çalışması beş adımdan oluşur:

1. Değiştirilmek istenen işlev için bir soyutlama oluşturulur.
2. Mevcut işlevi kullananlar, yeni soyutlamayı kullanacak şekilde değiştirilir.
3. Bu noktada, yeniden düzenlenmiş işlev ile soyutlamanın yeni uygulanan arayüzü oluşturulur. Bu yeni uygulanan arayüz, yeni mikroservise istekte bulunur.
4. Soyutlama yeni uygulamayı kullanacak şekilde değiştirilir.
5. Soyutlama temizlenir ve eski uygulama kaldırılır.



Şekil 3. Branch By Abstraction Örneği

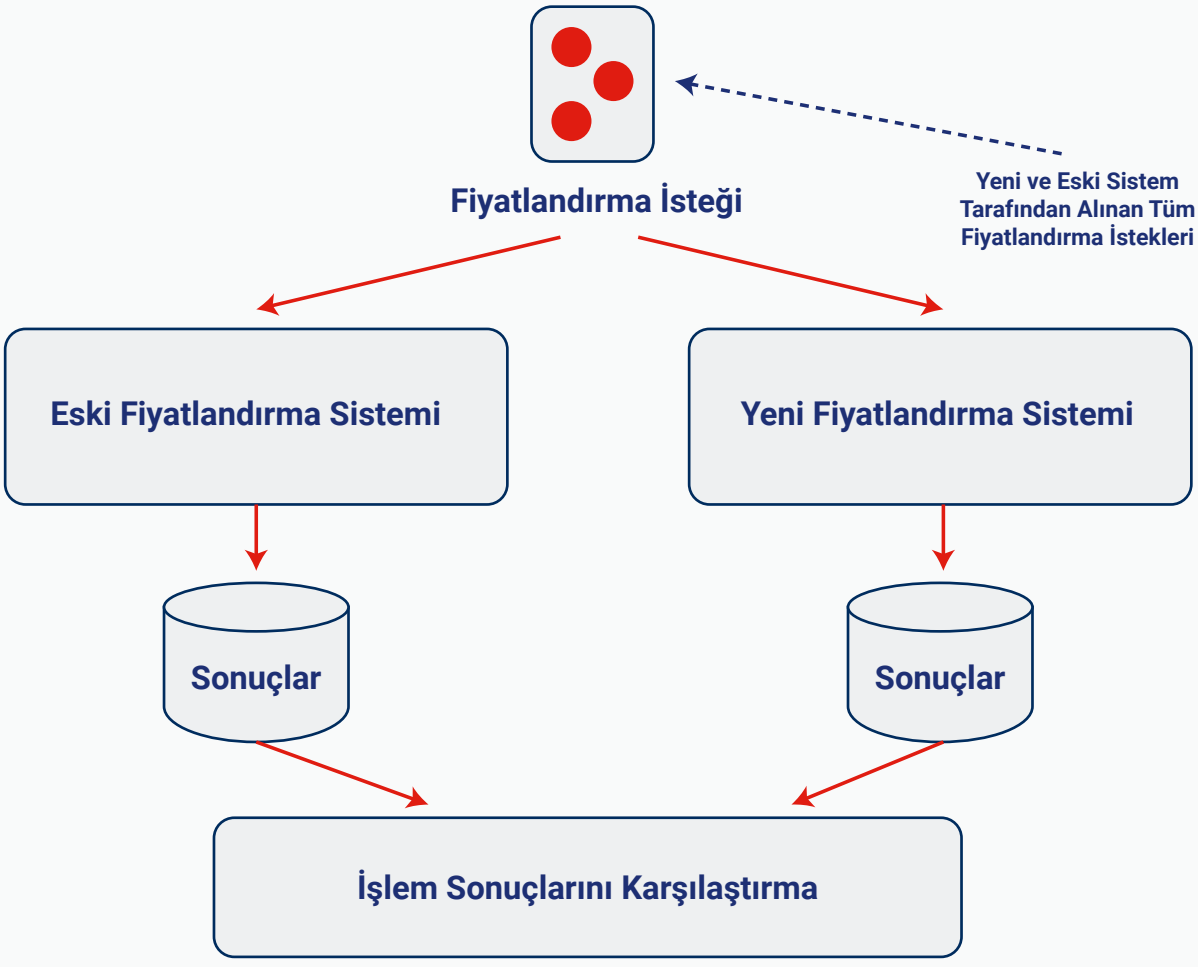
Branch by abstraction, geliştiricilerin fazla rahatsız edilmesi istenilmediği durumlarda kullanışlı bir desendir.

Bu desen ayrıca, mevcut sistemin kodunun değiştirilebileceğini varsayar.

1.1.3. Parallel Run

Hem strangler fig deseni hem de branch by abstraction deseni, aynı işlevin eski ve yeni uygulamalarının aynı anda canlı ortamda var olmasına olanak sağlar. Genellikle her iki teknikte, işlevin ya monolitik sistemdeki eski uygulamasına ya da yeni mikroservis tabanlı çözümünün yürütülmesine izin verir. Bu desenler yeni mikroservis tabanlı uygulamaya geçiş risklerini azaltmak için hızla önceki uygulamaya geri dönülmesine imkan sağlar.

Paralel Run kullanıldığında, eski veya yeni uygulamaya istek atmak yerine her ikisine de istek atılır, böylece sonuçlar karşılaştırılır ve eşit olduğundan emin olunur. Her iki uygulamaya istek atılmasına rağmen, herhangi bir anda sadece biri doğru kaynak olarak kabul edilir. Genellikle eski uygulama, devam eden yeni uygulamaya güvenilmesi gerektiğini gösterene kadar doğru kaynak olarak kabul edilir.



Şekil 4. Parallel Run Örneği

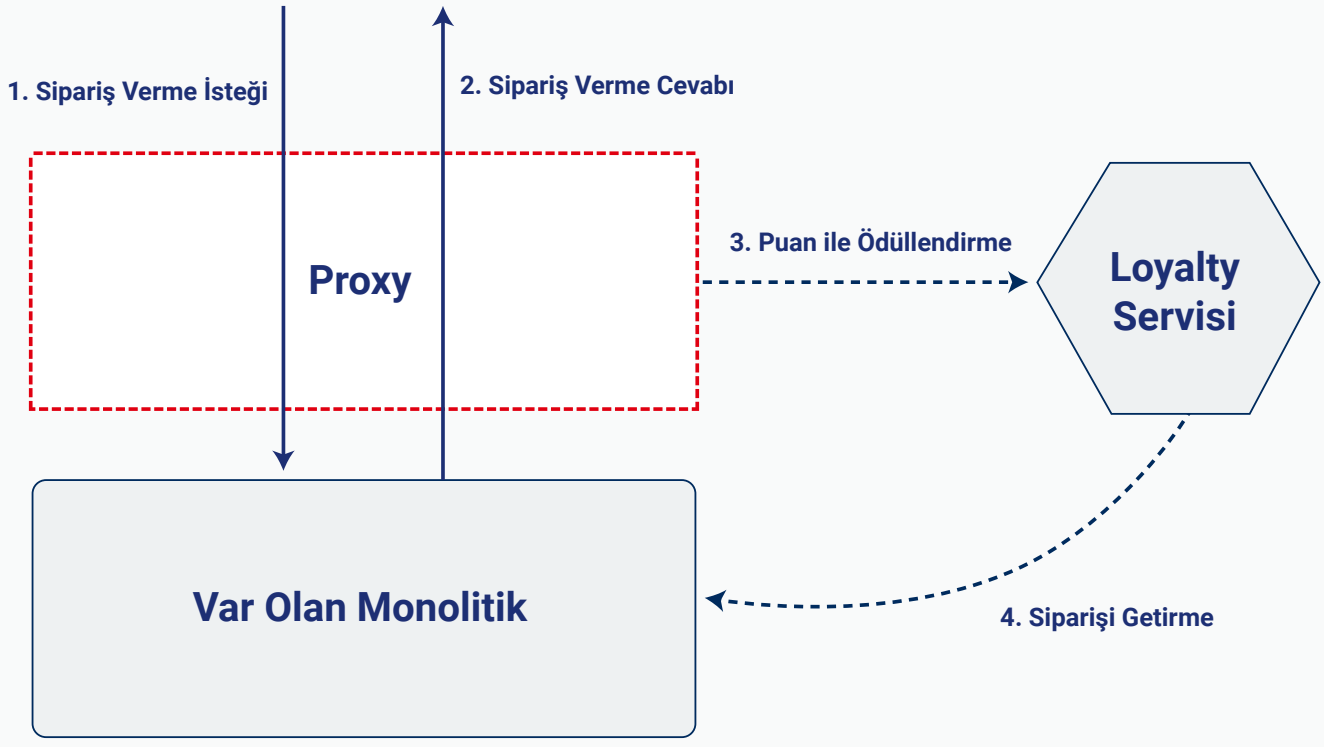
Bu teknik (Parallel Run), sadece yeni uygulamanın mevcut uygulama ile aynı cevapları verip vermediğini değil, aynı zamanda kabul edilebilir fonksiyonel olmayan parametreler içinde çalışıp çalışmadığını da doğrulamak için kullanılabilir. Örneğin "Yeni mikroservis yeterince hızlı yanıt veriyor mu, çok fazla zaman aşımı görünüyor mu?" gibi sorulara cevap verir.

1.1.4. Decorating Collaborator

Monolitik mimari proje içinde bir şeyin gerçekleşmesine dayalı olarak bazı davranışların tetiklenmesi isteniyor ama gerekli değişiklik yapılamıyorsa, decorating collaborator deseni burada büyük ölçüde yardımcı olabilir. Yaygın olarak bilinen decorator deseni, bir uygulama alt sistemi hakkında hiçbir bilgi sahip olmadan yeni işlevselliği eklemeye izin verir. Bu desen, monolitin altındaki yapı değiştirilmediğinden, monolitik sistemin doğrudan servislere istek yapmasını sağlıyormuş gibi görünmesini sağlamak için decorator kullanılmasına olanak tanır.

Bu istekleri monolitik sisteme ulaşmadan önce ele almak yerine, isteklerin normal olarak gerçekleşmesine izin verilir. Ardından, isteğin sonucuna bağlı olarak mikroservise bir vekil (proxy) aracılığıyla çağrı yapılabilir.

Bu yeni mikroservis, monolitik sistemden açığa çıkması gereken bilgileri kullanıp kullanmamak da dahil olmak üzere monolitin bilgilerini kullanabilir veya kullanmayabilir. Monolitik sistemden bilgi gerekiyorsa ama aynı zamanda monolitik sistemden istekler yapılıyorsa, döngüsel bağımlılık (circular dependency) olabileceğine dikkat edilmelidir.

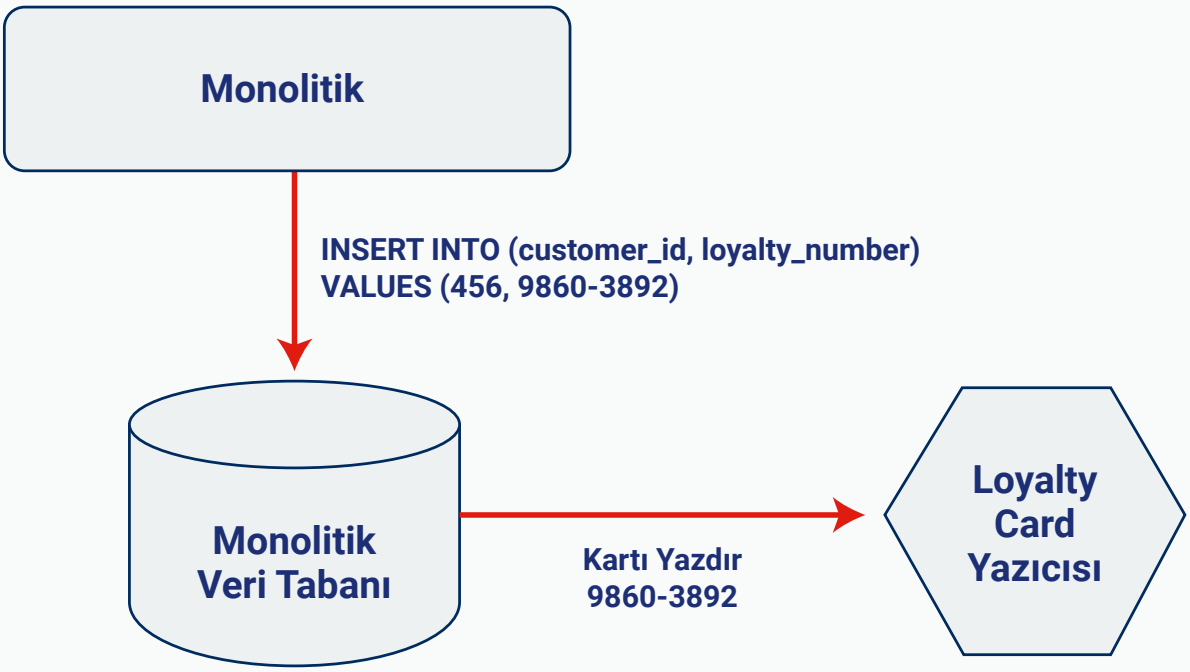


Şekil 5. Decorating Collaborator Örneği

Basit tutulduğunda bu desen, change data capture yöntemine göre daha iyi ve daha az bağlı (coupled) bir yaklaşımdır. Bu desen, gerekli bilginin gelen istekten veya monolitik sistemden gelen yanıttan çıkartılabildiği durumlarda en iyi şekilde çalışır. Ancak, yeni servise isteklerin yapılabilmesi için daha fazla bilgi gerekiyorsa, bu desenin kullanımının getireceği karmaşıklık ve sorunların düşünülmesi oldukça önemlidir.

1.1.5. Change Data Capture

Change data capture deseni ile, monolite yapılan çağrıları engellemeye ve üzerine işlem yapmaya çalışmak yerine, veri tabanında yapılan değişikliklere tepki verilir. Bu desenin kaçınılmaz zorluğu Change Data Capture işlemi için, temel yakalama sisteminin monolitin veri tabanına bağlı olmasıdır.



Şekil 6. Change Data Capture Örneği

Change Data Capture, özellikle verilerin çoğaltılması gerektiği durumda kullanışlı bir desendir.

Mikroservis geçişi sırasında, monolitik yapıda meydana gelen veri değişikliklerine uyum sağlama gerekliliği ortaya çıkabilir. Meydana gelen bu veri değişikliklerinin, strangler fig veya decorator desenleri kullanılarak algılanamadığı veya kod yapısının değiştirilemediği durumlarda bu yaklaşım oldukça faydalıdır.

Bu desen veri tabanı tetikleyicileri ve veri yakalama araçlarına ihtiyaç duyar. Bu gibi ekstra araçlara ihtiyaç duyması da çözüme karmaşıklık kattığından bu desen genellikle diğer yöntemlerin kullanılamadığı durumlarda kullanılır.

1.2. Monolitik Mimarideki Veri Tabanı Yapısının Ayrıştırılması

Monolitik uygulamaların kendilerine ait monolitik veri tabanları vardır. Mikroservis mimarisinin bir ilkesi, her mikroservis için bir veri tabanıdır. Bu nedenle, monolitik bir uygulama mikroservislere dönüştürülmek istendiğinde monolitik veri tabanı da bölünmek zorundadır.

İlk adım, monolitik veri tabanını analiz etmektir. Servis ayrıştırma sürecinde mikroservisler hakkında elde edilmiş bilgiler kullanılarak veri tabanı kullanımı analiz edilmeli ve tabloları veya diğer veri tabanı nesneleri yeni mikroservislere eşleşmelidir. Bu eşleme, potansiyel mikroservis sınırlarını aşan veri tabanı nesneleri arasındaki bağlantının anlaşılmasına yardımcı olur.

Monolitik bir veri tabanını farklı veri tabanlarına bölmek de karmaşık bir problemdir. Temelde monolitik bir veri tabanı monolitik uygulamanın bir yansımasıdır ve farklı işlevlere ait veri tabanı nesneleri arasında net bir ayrım yoktur. Örneğin, bir e-ticaret uygulamasında, ürün bilgileri, alışveriş sepeti verileri ve kullanıcı bilgilerini ayrı ayrı tablolarda düzenlenmediği durumda bu ayrımdan söz edilemez. Bunun yanında veri tabanlarını ayırırken veri senkronizasyonu, işlem bütünlüğü, gecikme ve birleştirme sorguları gibi noktalar da göz önünde bulundurulmalıdır.

Monolitik veri tabanını bölerken bu sorunlara nasıl yanıt verileceğini inceleyen çeşitli desenler vardır:

- Referans Tabloları (Reference Tables)
- Paylaşılan Referans, Sabit Veri (Shared Reference - Static Data)
- Paylaşılan Değişken Veri/Durum (Shared Mutable Data/State)
- Paylaşılan Tablolar (Shared Tables)

1.2.1. Referans Tabloları

Monolitik uygulamalarda en yaygın görülen desenlerden biri referans tablosudur. Bu desende, işlev veya modül, başka bir işlev veya modülün tablosuna erişir. Bir modülün kendi tablosu ile başka bir tablo arasındaki birleştirmeler, gereken veriyi almak için kullanılır. Bu, mikroservis mimarisinde anti-pattern olarak kabul edilir.

Veri tabanı nesnelerini ayırmak için düşünülmesi gereken iki seçenek vardır.

Seçenek 1: Veriyi API Olarak Kullanma

Temel işlevler veya modüller mikroservis olarak ayrıldığında, veriyi paylaşmak ve sunmak için yaygın bir yol API'lerdir. Referans servis, arayan servisin ihtiyacı olan veriyi API olarak sunar. Bu durumda birleştirmeler hafızada yapılır. Bu seçenek, veri boyutu sınırlı olduğunda çalışır, ancak ek ağ ve veri tabanı çağrılarına sahip olması nedeniyle performans sorunlarına yol açabilir. Ayrıca, hafızadaki veri kümesi birleştirmeleri artırır. Ancak bu durum, veri boyutu sınırlı olduğunda çalışır.

Seçenek 2: Verinin Projeksiyonu

İki ayrı mikroservis arasında veri paylaşmanın başka bir yolu, bağımlı serviste verinin bir projeksiyonunu oluşturmaktır. Veri projeksiyonu yalnızca okunur ve istendiğinde yeniden oluşturulabilir. Bu desen, servisin daha bütünsel olmasını sağlar.

1.2.2. Paylaşılan Referans-Sabit Veri

Ülke kodları, i18n ve desteklenen para birimleri gibi statik veriler, çok yavaş değişir ve genellikle bunları yönetmek için bir kullanıcı arayüzü mevcut değildir. Bu tür veriler, bir monolitik uygulamanın farklı işlevleri veya modülleri tarafından kendi varlıkları ile birleştirilerek erişilir. Bir mikroservis mimarisine geçişte, bu statik verilerle de başa çıkılması gerekir. Uygulama genelinde kullanılan statik veriler için aşağıdaki seçenekler düşünülebilir.

Seçenek 1: Veri Tabanına Göre Statik Veri

Farklı servislerin veya modüllerin uygulama içinde kullandığı statik veri, ayrı veri tabanlarına kopyalanabilirken ayrı mikroservislere modellenirler. Bu yaklaşımın dezavantajı, veri tutarlılığı ve veri çoğaltma sorunudur. Ancak bu yaklaşım, bu tür statik verilerin veri güncellemelerinin sık olmadığı durumlarda çok iyi çalışır.

Seçenek 2: Veri Olarak API Kullanma

Verileri API olarak kullanmanın standart bir yaklaşımı, paylaşılan referans verilerini bir alan olarak modellemek ve bunu yönetmek için ayrı bir mikroservis geliştirmektir. Bu mikroservis sahip olduğu veriyi, ihtiyaç duyan servislerin kullanabilmesini sağlayacak bir arayüz sağlar. Bu sayede diğer servisler ihtiyaç duydukları durumda arayüz sayesinde veriye erişebilir.

Seçenek 3: Statik Veri Olarak Yapılandırma

Bu yaklaşım, statik verilerin bir veri tabanında tutulması veya bir API aracılığıyla sunulması yerine, verilerin yapılandırma dosyaları veya yapılandırma ortamları aracılığıyla mikroservislere doğrudan eklenmesini içerir. Modern mikroservisler, bu tür yapılandırma verilerini yapılandırma sunucuları, anahtar-değer depolama sistemleri veya güvenli veri depolama mekanizmaları (vaults) gibi araçlar ile yönetmek için özellikler sunar. Bu özellikler bildirimsel olarak dahil edilebilir.

1.2.3. Paylaşılan Değişken Veri/Durum

Monolitik uygulamalarda, paylaşılan değişken durum (shared mutable state) olarak bilinen yaygın bir desen bulunmaktadır. Bu desen, farklı uygulama parçaları ve modüller tarafından erişilen, belirli etki alanı gerçekliklerini veri tabanında temsil eder. Temsil edilen bu bilgi birçok işlev tarafından kullanılan ve devamlı değişebilen etki alanı bilgisidir. Bu çoğunlukla kolaylık için yapılır.

Mikroservis mimarisi geçişi sırasında aynı tablo üzerinde çalışan servisleri bulmak zordur. Ancak verinin tek bir sahibi olması gerekmektedir. Bu durumda verinin sahibine karar verilir ve verinin sahibi bir API sunarak diğer servislerin de veriyi kullanabilmesini sağlar.

Örneğin müşterilerin sipariş durum (order status) bilgisinin tutulduğu “alisveris_durumu” tablosu Sipariş, Ödeme ve Kargo işlevleri tarafından kullanılmaktadır. Değişen sipariş durumu bilgisinin

tutulduğu "alisveris_durumu" tablosu, tabloyu yöneten Alışveriş Durumu mikroservisine çıkılır. Alışveriş Durumu servisinin sahip olduğu API aracılığı ile Sipariş, Ödeme ve Kargo mikroservisleri "alisveris_durumu" tablosunda tutulan bilgiye erişim sağlayacaktır.

1.2.4. Paylaşılan Tablolar

Paylaşılan tablolar deseni, paylaşılan değişken veri/durum deseni ile çok benzerdir ve hatalı etki alanı modellemenin sonucudur. Bu senaryoda, veri tabanı tablosunun, birden fazla farklı işlev veya modül tarafından gereksinim duyulan özellikleri içerecek şekilde modellemesi yapılır. Bu da genellikle kolaylık nedeniyle yapılır.

Burada, ayrılacak mikroservisler göz önüne alınarak oluşturulan sınırlı bağlamlara bakılarak tablodaki nesneler (entity) ayrılır.

Örneğin; ürün ve stok işlevleri "urun" tablosunu ortak olarak kullanmakta iken, ürün ve stok ayrı servislere çıkılması durumunda, "urun" tablosunda yer alan alanlar çıkılacak olan mikroservislerdeki alanlara göre ayrılır. Ürün mikroservisinin kullanacağı tablo "urun" olarak kalırken, stok servisinin kullanacağı tablo "urun_stok" olarak ele alınır.

1.3. İşlem (Transaction) Bütünlüğü Yönetiminin Ele Alınması

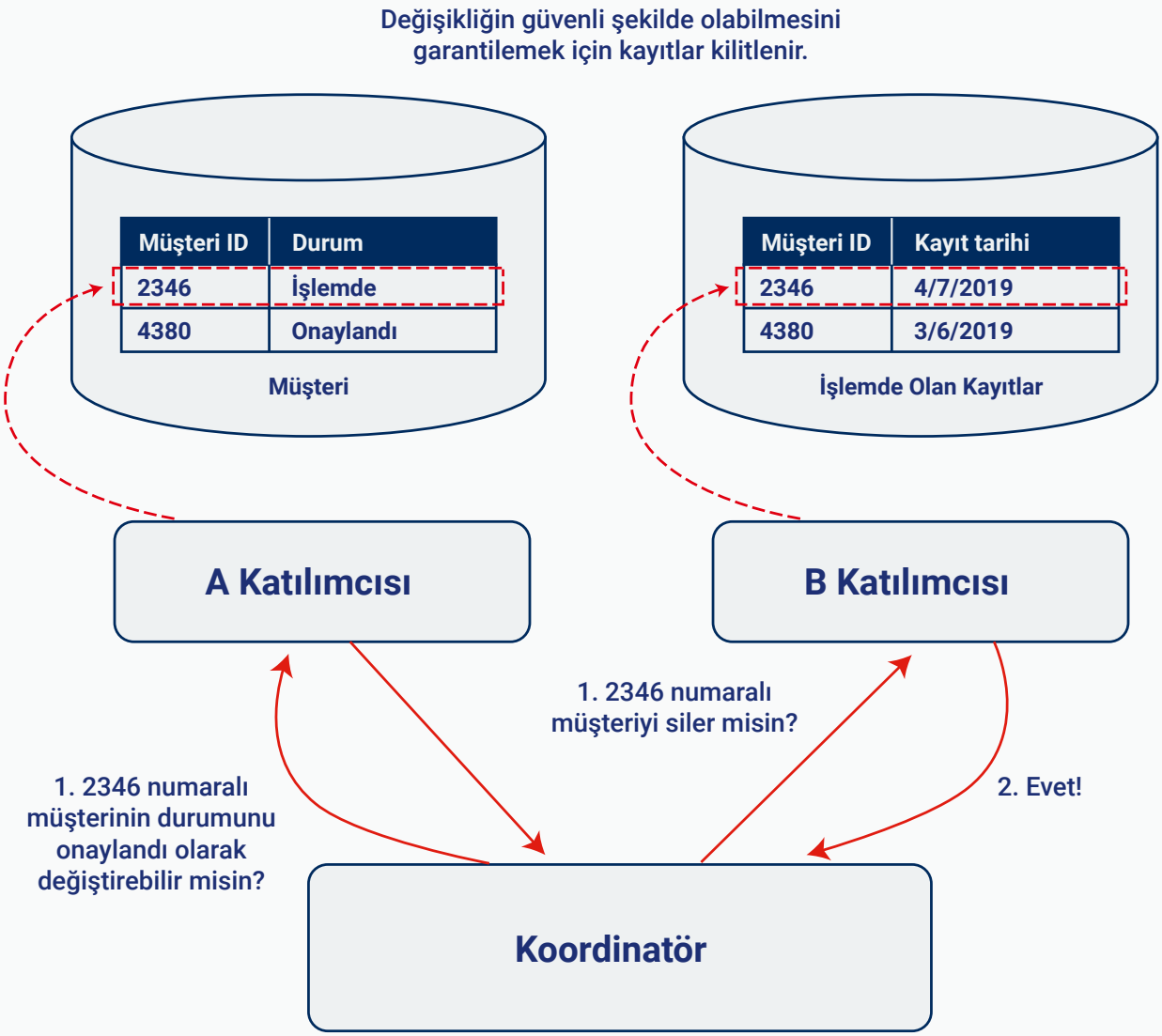
Veri yönetimi her uygulamanın önemli bir parçasıdır. Mikroservisler doğası gereği her servis için farklı kategorilerde veri tabanı sağlayıcılarının kullanılması özgürlüğünü getirir. Servis başına veri tabanı olarak adlandırılan bu model, zorlukları da beraberinde getirir. Monolitik uygulamalarda geleneksel veri tabanı kullanıldığından ACID işlemlerini kullanmak kolaydır. Fakat her biri kendi veri tabanına sahip birden fazla mikroservis olduğunda, işlemleri ele almak daha zordur ve işlemlerle uğraşmak için daha fazla zaman harcanması gerekir. Çünkü her bir mikroservis, kendi veri tabanını yönetir ve belirli bir işlevi yerine getirir. Bu, işlem bağımsızlığı anlamına gelir. Örneğin; bir e-ticaret uygulamasında sipariş işlemleri farklı bir mikroservis tarafından yönetilirken, ödeme işlemleri başka bir mikroservis tarafından ele alınabilir.

Mikroservislerdeki veri tabanı bağımsızlığı ise her mikroservisin kendi veri tabanını yönettiği bir yapıya işaret eder. Her mikroservis, kendi veri tabanı üzerinde işlem yapar ve bu veri tabanı diğer servislerle paylaşılmaz veya otomatik olarak senkronize edilmez. Bu, her mikroservisin kendi verilerini bağımsızca kontrol etmesine olanak tanır, ancak diğer servisler tarafından gerçekleştirilen işlemlerin sonuçlarını otomatik olarak bilemezler.

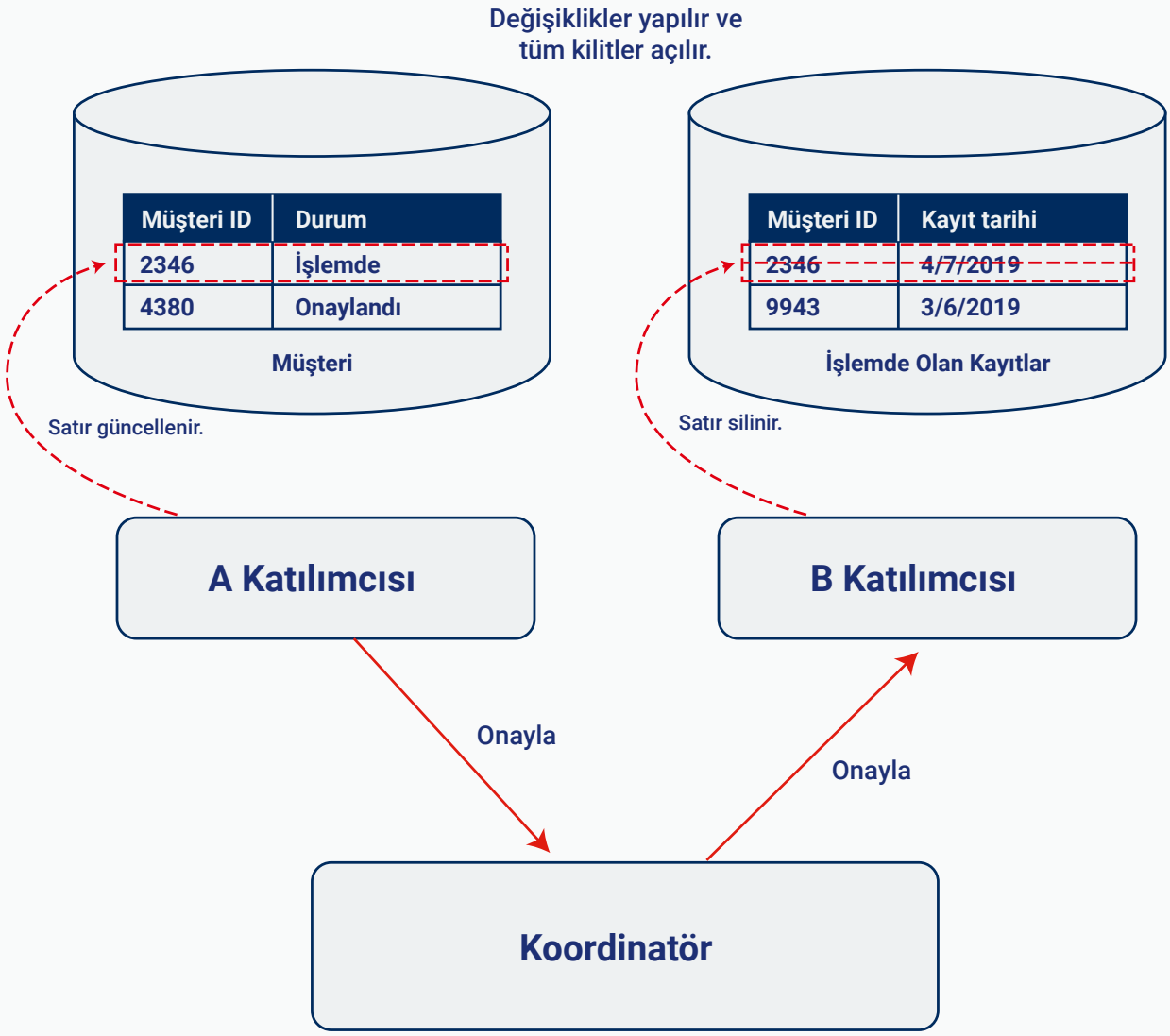
İşlem adımlarının her biri farklı bir mikroservis ve onun veri tabanı içinde çalıştığından, tüm işlem için ACID ilkesini korumak son derece zor ve karmaşıktır. Bu durumlarda mümkünse dağıtık işlem bütünlüğü yönetiminden kaçınmak daha iyidir. Fakat bu kaçınılmazsa asenkron ve senkron olmak üzere iki tür işlem bütünlüğü yönetim deseni mevcuttur. Senkron için en çok bilinen Two-Phase Commit, asenkron içinse Saga Pattern örnek verilebilir.

1.3.1. Two-Phase Commits

Two-Phase Commit algoritması (bazen 2PC olarak kısaltılır), dağıtık bir sistemde işlem yapma yeteneği sağlamak için sıkça kullanılır. Bu tür sistemlerde birden fazla ayrı işlem, genel işlemin bir parçası olarak güncellenmesi gerekebilir. Koordinatör düğüm (coordinator node) ve birden fazla katılımcı düğüm (participant node) arasında geçen bu işlem iki aşamaya ayrılır: Oylama aşaması (voting phase) ve onaylama aşaması (commit phase). Oylama aşamasında, koordinatör düğüm işlemin bir parçası olacak tüm katılımcı düğümler ile iletişime geçer ve durum (state) değişikliklerinin yapılıp yapılamayacağı konusunda onay ister. Eğer tüm katılımcı düğümler, kendilerinden istenen durum değişikliğinin gerçekleşebileceğini onaylarsa, işlem onaylanır (transaction commit). Eğer herhangi bir katılımcı "düğüm" değişikliğinin gerçekleşmeyeceğini belirtirse, tüm işlem iptal edilir (transaction rollback).



Şekil 7. Voting Phase Örneği



Şekil 8. Commit Örneği

2PC, tüm düğümlerin bir işlemi onaylayacağını veya iptal edeceğini garanti eder, ancak bu işlem yavaş çalışabilir ve başarısızlığa karşı savunmasız olabilir.

1.3.2. SAGA

Two-phase commit'in aksine, saga birden çok durum değişikliğini koordine edebilen bir algoritmadır. Saga, yerel işlemler (local transaction) dizisidir. Saga içindeki her bir servis kendi işlemini gerçekleştirir ve bir olay (event) yayınlar. Diğer servisler bu olayı dinler ve bir sonraki işlem (transaction) bütünlüğünü gerçekleştirir. Bir işlem herhangi bir nedenle başarısız olursa, saga, önceki işlemlerin etkisini geri almak için telafi işlemlerini de yürütür.

Saga deseni, birden çok servis arasında veri tutarlılığını sağlamaya yardımcı olurken, temel olarak atomikliği garanti etmez.

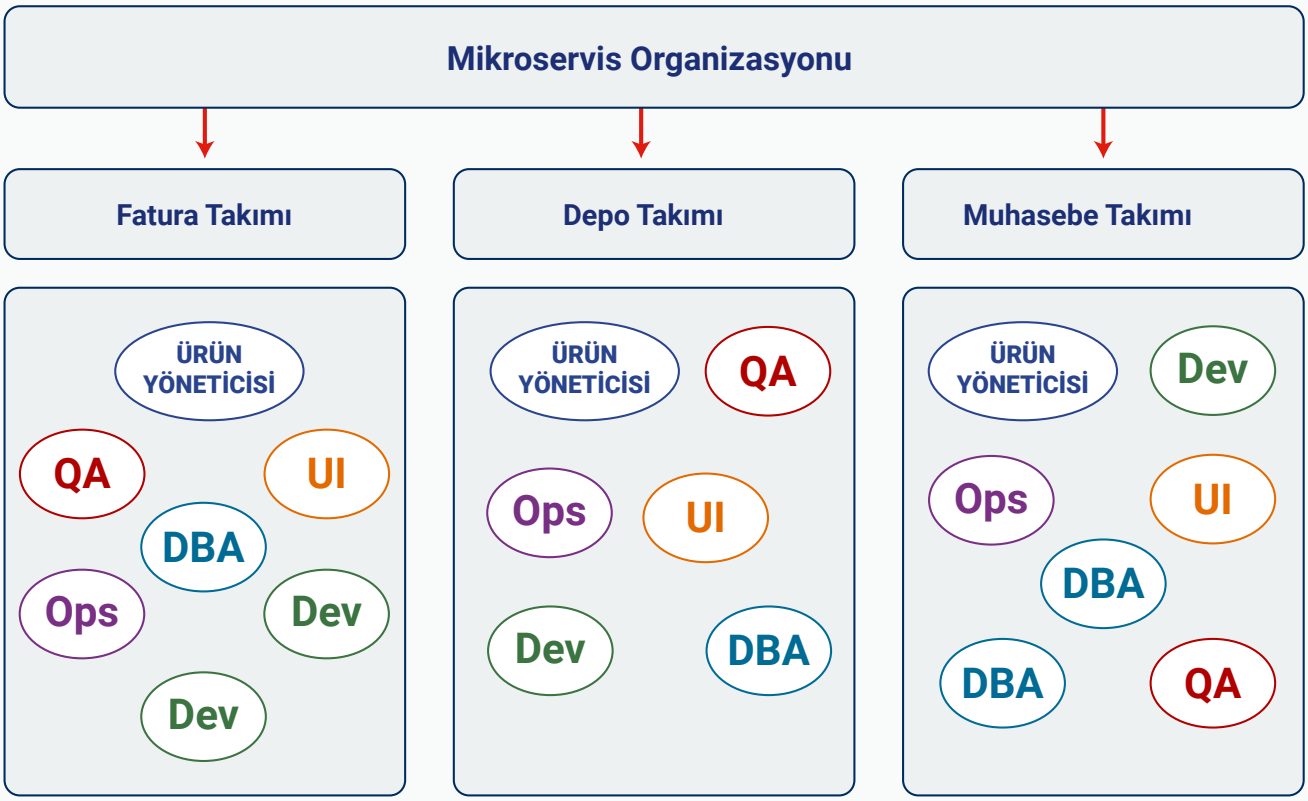
Organizasyonel Zorluklar

Teknik zorlukların yanında küçümsenmemesi gereken ve bir o kadar da önemli organizasyonel zorluklar mevcuttur.

Organizasyonel zorluklardan biri de organizasyonun yapısıdır. İyi bir uygulama geliştirmek için organizasyonun kendi yapısını uygulama mimarisinin yapısıyla uyumlu hale getirmesi gerekir (Newman, 2014). Eğer daha önce monolitik uygulamayı kalite güvence, geliştirme, veri tabanı yönetimi gibi net rolleri olan büyük ekipler ele almışsa, o zaman bu tür bir organizasyon yapısı mikroservislerle çalışmaz. Conway' in yasasına göre sistemi tasarlayan organizasyon, organizasyon yapısının kopyası gibi olan bir sistem üretecektir (Conway, M. E. 1968). Eğer organizasyonun yapısı monolitik ise mikroservis yaklaşımı çalışması beklenmez. Bu durumda organizasyon, büyük ekipleri özerk olarak çalışabilecek daha küçük ekiplere bölmelidir.

Şekil 9. Monolitik Organizasyonda Monolitik Uygulama Teslimatı

Şekil 9, alanlarında uzman ekiplerden oluşan monolitik bir organizasyonu göstermektedir. Her bir ekip uzmanlık alanları açısından bakıldığında çok iyilerdir, fakat iş teslimi sırasında birbirleriyle işbirliği yapmaları gerekmektedir. Böyle bir yapıda sürüm yayınlamadan önce yapılacak olan değişikliklerin test edilmesi, onaylanması ve uygun olması durumunda kullanıma sunulması gibi devir süreçleri bulunmaktadır (Mauro, 2015). Bu türden bir yapı yavaş döngülere sebep olmakta ve iş tesliminde aksaklıklara sebep olmaktadır.



Şekil 10. Mikroservis Mimari Tarafından Desteklenen Organizasyon Yapısı

Şekil 10, mikroservisler ve ürünler etrafında yapılandırılmış bir organizasyonu göstermektedir. Ekipler bu şekilde organize olduklarında, sürümler göz önünde bulundurularak daha fazla özerkliğe sahip olurlar. Artık üçüncü bir tarafa devir süreci yoktur ve takımların diğer takımların değişikliklerini tamamlamasını beklemesine gerek yoktur (Mäkitalo, 2018).

Mikroservis mimariyi uygulayabilmek için en önemli şeylerden birisi de DevOps kültürünün benimsendiği ve uygulandığı ekipler oluşturabilmektir. Martin Fowler, yazdığı makalelerde mikroservislerin sağladığı operasyonel esneklik ve dağıtık sistemlerin yönetiminde DevOps prensiplerinin önemine değinir. Öncelikle, servislerin yapılandırmalar, konfigürasyon ve dağıtım süreçleri için bir dağıtım/teslimat hattının tasarlanması ve hayata geçirilmesi gerekmektedir. Bir sistemde değişiklik yapılması ile değişikliğin gerçekleştirilmesi arasındaki sürenin azaltılması yüksek kaliteyi sağlar (Bass, Weber, Zhu, 2015). Dağıtımların hızlı sorunsuz olması mikroservislerin önemli noktalarından biridir. Bu tür bir dağıtım sürecine sürekli dağıtım/teslimat denir (Wettinger J., Andrikopoulos V., Leymann F., 2015). Monolitik bir organizasyonda geliştiriciler sadece kod yazar ve dağıtım (deployment) aşamasını operasyon takımına bırakırlar. Mikroservis mimari ile birlikte her ekibin dağıtımları idare edebilmesi gerekmektedir. Bu durumdan dolayı ekip üyelerinin dağıtım süreçleri hakkında yeni beceriler öğrenmesi ve kendilerini geliştirmeleri gerekmektedir. Bu durumda organizasyon ve içinde çalışan elemanlar için zaman ve sabır gerektirir (Mäkitalo, 2018).

Sonuç ve Öneriler

Monolitik mimariden mikroservis mimariye geçiş kolay değildir ve organizasyonun ve projenin çeşitli bölümlerinden çok fazla zaman, çaba ve maliyet gerektirir. Bir sistemi dağıtık hale getirmek, ele alınması gereken yeni zorlukları ortaya çıkarır. Dolayısıyla, mikroservislerde başarılı olmak için teknik ve organizasyonel zorlukların her ikisinin de çözülmesi gerekir. Mikroservislere geçiş uzun zaman alabilecek büyük bir süreçtir ve bu geçişi düşünen organizasyonlar, geçiş için sahip oldukları gerekçeleri iyi analiz etmeli, geçişin maliyetini ve getirisini değerlendirmeli ve kendi sorunlarını göz önünde bulundurmalıdır.

Monolitik mimariden mikroservis mimarisine geçiş, büyük bir çaba ve özen gerektiren karmaşık bir süreçtir. Bu dönüşümün organizasyonunun ve projesinin farklı yönlerinden önemli zaman, emek ve maliyet gerektirdiği görülür. Yeni bir dağıtık sistem oluşturmak, yeni zorlukların üstesinden gelmeyi gerektirirken, teknik ve organizasyonel engellerin her ikisinin de aşılması gerekir.

Mikroservis mimarisine geçiş, sadece teknik bir dönüşüm değil, aynı zamanda organizasyonun kültürel ve iş süreçlerini de değiştirir. Bu nedenle, başarılı bir geçiş için tüm paydaşların katılımı ve destekleri hayati önem taşır. Aynı zamanda, bu sürecin uzun vadeli bir vizyon ve strateji ile desteklenmesi gereklidir.

Mikroservislerin sunduğu esneklik, ölçeklenebilirlik ve hız gibi avantajlar göz önüne alındığında, bu dönüşümün potansiyel getirileri büyük olabilir. Ancak, bu geçişin maliyetini ve getirisini dikkatlice değerlendirmek önemlidir. Her organizasyonun ihtiyaçları farklıdır ve bu nedenle mikroservis mimarisine geçiş kararı özenle ele alınmalıdır.

Sonuç olarak, mikroservis mimarisine geçiş, zorlu bir süreç olabilir, ancak doğru bir şekilde planlandığında ve yönetildiğinde büyük faydalar sağlayabilir. Bu dönüşümü düşünen organizasyonlar, kendi özel koşullarını ve hedeflerini göz önünde bulundurarak kararlarını vermelidirler.

Kaynakça

1. Freddy Tapia, Miguel Ángel Mora, Walter Fuertes, Hernán Aules, Edwin Flores and Theofilos Toulkeridis, From Monolithic Systems to Microservices: A Comparative Study of Performance. 2020.
2. Challenges When Moving from Monolith to Microservice Architecture Miika Kalske, Niko M¨akitalo, and Tommi Mikkonen, February 2018.
3. Conway, M. E. (1968). How do committees invent. Datamation, 14(4), 28-31.
4. Hasselbring, W. (2016, March). Microservices for scalability: keynote talk abstract.
5. In Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (pp. 133-134). ACM.
6. Tony Mauro, (March 2015) [Online] https://www.nginx.com/blog/adopting-microservices-at-netflix-lessons-for-team-and-process-design/
7. Wettinger J., Andrikopoulos V., Leymann F. (2015) Enabling DevOps Collaboration and Continuous Delivery Using Diverse Application Environments. In: Debruyne
8. C. et al. (eds) On the Move to Meaningful Internet Systems: OTM 2015 Conferences. Lecture Notes in Computer Science, vol 9415. Springer, Cham
9. L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.
10. Sam Newman, Building Microservices, Designing Fine-Grained Systems, 1st ed.United States of America: O'Reilly Media Inc., 2015.
11. Sam Newman, Monolith to Microservices, Evolutionary Patterns to Transform Your Monolith, 1st ed.United States of America: O'Reilly Media Inc., 2019.
12. https://developer.ibm.com/articles/challenges-and-patterns-for-modernizing-a-monolithic-application- into-microservices/
13. https://medium.com/javarevisited/distributed-transaction-management-in-microservices-part-1-bb7dc1fbee9f#:~:text=Transactions%20in%20Monolith%20and%20Microservices&text=In%20the%-20case%20of%20Microservices,the%20other%20service%27s%20database%20directly.
14. https://microservice-api-patterns.org/patterns/responsibility/informationHolderEndpointTypes/ReferenceDataHolder.html
15. https://auth0.com/blog/introduction-to-microservices-part-4-dependencies/
16. https://github.com/suadev/turkish-microservice-architecture-book/blob/master/Mikroservis_Mimari_v1.0.pdf
17. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20application%20is%20built,on%20a%20number%20of%20factors.
18. https://developer.ibm.com/articles/use-saga-to-solve-distributed-transaction-management-problems-in-a-microservices-architecture/
19. https://martinfowler.com/articles/microservice-trade-offs.html

T.C. SANAYİ VE
TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

İşçi Blokları Mahallesi Muhsin Yazıcıoğlu Caddesi No:51/C 06530 Çankaya/ANKARA

+90 (312) 289 92 22 - yte.bilgi@tubitak.gov.tr

TÜBİTAK - BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE)