T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BÜYÜK ÖLÇEKLİ UYGULAMALARIN DEVOPS SÜREÇLERİNE UYARLANMASI VE UYGULANMASI

YÜKSEK LİSANS TEZİ

Ömer Furkan ARİFOĞLU

Enstitü Anabilim Dalı : BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ

Tez Danışmanı : Prof. Dr. Ümit KOCABIÇAK

Ocak 2020

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BÜYÜK ÖLÇEKLİ UYGULAMALARIN DEVOPS SÜREÇLERİNE UYARLANMASI VE UYGULANMASI

YÜKSEK LİSANS TEZİ

Ömer Furkan ARİFOĞLU

Enstitü Anabilim Dalı : BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ

Bu tez 28.01.2020 tarihinde aşağıdaki jüri tarafından oybirliği / oyçokluğu ile kabul edilmiştir.

Prof. Dr.	**Dr. Öğr. Üyesi**	**Dr. Öğr. Üyesi**
Ümit KOCABIÇAK	**Mustafa Zahid YILDIZ**	**Veysel Harun ŞAHİN**
Jüri Başkanı	**Üye**	**Üye**

BEYAN

Tez içindeki tüm verilerin akademik kurallar çerçevesinde tarafımdan elde edildiğini, görsel ve yazılı tüm bilgi ve sonuçların akademik ve etik kurallara uygun şekilde sunulduğunu, kullanılan verilerde herhangi bir tahrifat yapılmadığını, başkalarının eserlerinden yararlanılması durumunda bilimsel normlara uygun olarak atıfta bulunulduğunu, tezde yer alan verilerin bu üniversite veya başka bir üniversitede herhangi bir tez çalışmasında kullanılmadığını beyan ederim.

Ömer Furkan ARİFOĞLU
28.01.2020

TEŞEKKÜR

Yüksek lisans eğitimim boyunca değerli bilgi ve deneyimlerinden yararlandığım, her konuda bilgi ve desteğini almaktan çekinmediğim, araştırmanın planlanmasından yazılmasına kadar tüm aşamalarında yardımlarını esirgemeyen, teşvik eden, aynı titizlikte beni yönlendiren değerli danışman hocam Prof. Dr. Ümit KOCABIÇAK'a teşekkürlerimi sunarım.

İÇİNDEKİLER

SİMGELER VE KISALTMALAR LİSTESİ

BT	: Bilgi Teknolojileri
CAST	: Center for Autonomous Systems and Technologies
DevOps	: Development Operations
DSDM	: Dynamic Software Development Methodology
HoC	: Higher order Component
Java EE	: Java Enterprise Edition
SDLC	: Software Development Life Cycle
XP	: Extreme Programming

ŞEKİLLER LİSTESİ

ÖZET

Anahtar kelimeler: Eski yapılı sistemler, DevOps, Uygulama Geliştirme Metodolojisi, Şelale Model, Çevik Model, Uygulama Geliştirme Yaşam Döngüsü, Uygulama geliştirme, Yazılım geliştirme

Bu çalışmada, eski proje geliştirme modellerini kullanarak sahip oldukları iş akışlarına ve ihtiyaçlarına uygun yazılım projeleri geliştirmiş kurum ve kuruluşların, uzun süredir sahip oldukları ve günümüzde de kullandıkları uygulama geliştirme modellerinin tanımlamalarına değinilmiştir. Yapılan bu tanımlamaların ardından zaman içinde meydana gelen ihtiyaçlar sonucu gerekli değişikliklerin ve DevOps'a entegrasyonun neden ve sonuçlarını ele alarak, kuruluşların yaşadıkları DevOps geçiş tecrübeleri, uygulamalı örnekleri ile anlatılmıştır.

Kullanım olarak eskide kalmış ve günümüz uygulama geliştirme ihtiyaçlarına cevap veremeyen yazılım geliştirme modelleri ve proje yaşam döngüleri, ilk kullanılmaya başladığı yıldan bugüne kadar çeşitli nedenlerle değişikliklere uğramıştır. Bu gelişim olarak adlandırabileceğimiz değişimler farklı sebeplerin ve ihtiyaçların bir araya gelmesi ile oluşturulmuştur. Literatürde en eski olarak adlandırabileceğimiz yazılım geliştirme disiplin modeli Şelale modeli olarak geçmektedir. Daha sonrasında oluşan ihtiyaçlara göre bu model kronolojik olarak V-Şekilli model, Spiral model, Eklemeli model ve Çevik model olarak isimlendirilerek çeşitli yeniliklere uğramıştır. Günümüzde kullanım ve ihtiyaçlara cevap vermesi bakımından karşılaştırıldığında en verimli model, Çevik model olarak görülmektedir. Bunun sebebi gerek müşteri gerekse iş modeline göre disiplin anlayışını esnek tutabilmesidir. Diğer bir söylem ile tüm modellerin bir arada görülebildiği bir proje yönetim disiplini olarak günümüz yazılım geliştirme uygulamalarının ihtiyaçlarına cevap verebilmektedir. Sahip olduğu bu özelliklere rağmen Çevik model, çeşitli kuruluşlarca tercih edilmemektedir. Bunun en temel sebebi, bu tercihten kaçınan kuruluşların sahip oldukları eski yapılı sistemlerde yaşanan dönüşüm zorluklarıdır. Bu eski yapılı sistemlere sahip kurumların günümüzde hizmet verdikleri uygulamaları ve uygulamalarını geliştirmek adına izledikleri proje disiplinleri bu anlamda gerekli esnekliğe sahip değildir. Bunun yanında kuruluşların sahip oldukları kurumsal disiplin de ayrıca yeni ve daha esnek bir modele geçiş için açık çalışma anlayışına sahip değildir. Temelde bu iki ana sebebe sahip, gelişimi engelleyen faktörlerin yanında bahsedilen bu kuruluşların gelişimini engelleyen farklı sebeplerde mevcuttur. Araştırmada elde edilen bulgular, kurumsal düzeyde görülen bu unsurları ortaya koymaktadır. Ortaya konan bulgular aynı zamanda örnekler ile pekiştirilmiş ve sonuç olarak hali hazırda çalışmakta olan bir kuruluş üzerinde uygulanmıştır.

ADAPTATION AND INTEGRATION OF LARGE SCALE APPLICATIONS TO DEVOPS ENVIRONMENT

SUMMARY

Keywords: Legacy Systems, DevOps, Software Development Methodology, Waterfall Model, Agile Model, Software Development Life Cycle, Software Development

In this study, old school software development models that have been used for a long time period and still being used by organizations are defined. After these definitions, necessary changes and DevOps integrations that have occurred over time for these models and experiences of organizations which made a transition from old Project development models to DevOps are explained with practical examples by considering the causes and consequences.

The software development discipline model, which we can call the oldest in the literature is called as Waterfall Model. According to the needs that emerged later, this model has been renamed as V-Shaped model, Spiral Model, Additive Model and Agile Model chronologically. Today, the most efficient model is considered as the Agile model when compared in terms of usage and responding to needs. The reason for this is that it can keep the discipline flexible according to the customer and business model. In other words, as a project management discipline, Agile Model can meet the needs of today's software development applications. Despite these features, Agile model is not preferred by various organizations. The main reason for this is the transformation difficulties experienced in the old built systems of the organizations that avoid this preference. The project disciplines that institutions with these old-fashioned systems are pursuing to imporve the practices and applicatiaons they serve today do not have the necessary flexibility in this sense. In addition, the institutional discipline owned by organizations also does not have a clear understanding of working towards a new and more flexible model. Besides these factors that prevent development, there are different reasons that prevent the development of these institutions. Findings obtained in the research reveal these elements at institutional level. The findings also reinforced and consequently applied to an organization that has already an active business cycle.

BÖLÜM 1. GİRİŞ

Teknoloji, dünden bugüne ve bugünden yarına toplumda yaşayışı kolaylaştıran, bu uğurda oluşan talepler üzerine oluşturulmuş modernleşme sürecine bugüne kadar katkıda bulunmuş, toplumun gerek üretim gerek ise tüketim evrelerinde ilerlemesini sağlamış bir araçtır. Geçmişte üretim süreçlerinin hızlanma aşamalarında rol oynayan teknoloji, tarihin bir parçasında kendini üretim hatlarının bantlar üzerinde ilerlemesi olarak (fordizm) gösterirken, daha sonrasında üretimin farklı ülkelere taşınmasına ancak geliri üretim araçlarına sahip olan ülkelere aktarmasını sağlayan bir geçiş sürecini tamamlamıştır (Toyotizm). Günümüzde ise teknoloji üretim araçları ile sınırlı kalmayarak kendisini toplumun tüketim yapısındaki değişimler ile de göstermeye başlamıştır. Kronolojik olarak incelendiğinde, buharlı üretim araçlarının kullanımı, hareket eden bant üzerinden yapılan seri üretimler, jet motorları, ilk bilgisayar yazılımı kodları, ilk kişisel bilgisayarın ortaya çıkması, ilk akıllı telefon gibi nice devrim olarak tanımlanan buluşların aslında tarihinin çok da geçmişinde olmadığını görebiliriz. Gelişim ve evrim, günümüzde artık metalin enerji ile buluşması yolu ile değil, görmediğimiz rakamların görmediğimiz bir alanda oluşturduğu dizilimlerle tanımlanmaya başlamıştır.

Teknolojinin bu geçiş evresinde kullanışını ve verimliliğini yitiren birtakım yapılar ve sistemler yenilenmeye ve çağa ayak uydurmaya mecburdurlar. Gelişimin ve iş akışındaki farklı süreçlerin aynı anda çalışarak en verimli hale ulaşması ortak amacını gütmesi olarak tasvir edilecek olan DevOps (Development Operations) proje metodolojisi disiplini, bugün ile yakın geleceğin güncel konusu olma yolunda ilerlemektedir. Kısaca bahsetmek gerekirse DevOps, geliştirme ve operasyon ekiplerinin bir yazılım projesinin test, dağıtım ve yönetim aşamalarının hepsinin kapsandığı, SDLC'nin tamamında birlikte çalıştığı bir proje yönetim disiplinidir. Bu disiplinde isminden de anlaşılacağı üzere uygulama geliştiricileri (Developers)

operasyon yöneticileri (Operations) ile birleşerek projeleri farklı bakış açılarından görülebilen ihtiyaçlar çerçevesinde ilerletmektedirler (Garinchaud, 2012). Diğer bir söylem ile DevOps, proje oluşumunun, yalnızca projenin en başında bir kere kurgulanması yerine, projenin her bir aşamasında ve müşterinin talepleri ile yaptığı sürekli buluşmalar sırasında, varlığını henüz göremediğimiz problemlerin farklı çözümlerinin projeye işlenebileceği bir disiplin anlayışı ile çalışmaktadır. Ancak her geçiş evresinde görülebileceği üzere bahsedilen bu geçişin de beraberinde getirdiği birtakım zorluklar mevcuttur. Özellikle geçmişte kalan proje gelişim modellerini, kurumsal içyapılarının değişmez parçaları haline getirmiş, yoğun uygulama birikimine sahip kurum kültürlerinde bu zorlukların yüksek derecelere ulaştığı sıklıkla karşılaşılmaktadır (Bradley, 2015; Saran, 2015).

Bu araştırmanın amacı, belirtildiği gibi eski sistemlere sahip ve büyük uygulama yapıları olan kurum ve kuruluşların bulundukları eski sistemlerden DevOps sistemlerine geçişlerinde yaşadıkları sorunları incelemek ve gerçek örneklerin uyarlamaları ile ortaya koymaktır. Bu konu hakkında sorulması gereken soru, DevOps modelini özellikle eski sistemlere sahip ve iş akışlarında büyük uygulamalar kullanan kurumlarda uygulamanın zorlukları nelerdir. Diğer bir söylem ile her şirket veya kurum, sistemsel olarak gerçekleştirmesi gerektiği gelişim ve geçiş dönemlerinde birtakım hatalar ve zorluklar ile karşılaşabilir. Ancak bu tez ile özel olarak cevaplanmaya çalışılan problem, DevOps gelişim ve geçiş evresini hali hazırda hizmet veren birimlere uygulanmasında yaşanan sorunları göz önüne sermek ve yaşanan bu sorunları yaşanmış örnekler ile uygulamalı olarak ortaya çıkarmaktır. Ayrıca bahsedilen DevOps geçişlerine konu olan projeler yardımı ile hataların ve faydaların kendini en çok gösterdiği alanların incelemesini yapmaktır.

Bu araştırmayı gerçekleştirmek adına izlenecek yol, öncelikle teknolojinin evrimsel sürecini minimal bir seviyeye getirerek daha mikro bir ölçekte yazılım projeleri geliştirme metodolojilerine indirgemektir. Geçmişten bugüne piyasada kullanılan çeşitli gelişim modelleri bulunmaktadır. DevOps geçişini anlamayı kolaylaştırmak adına her bir modele kısaca yer verilmesi gerekmektedir. Ancak sahip olduğu

manifesto ile DevOps disiplinine en uygun model Çevik proje yaşam döngüsü modelidir. Bu çalışmada gerçekleştirilecek olan temel kavram tanımlamalarında bu uygunluk karşılaştırmalı olarak gösterilecektir. Kronolojik olarak sıralanan metodolojiler ve modellerin tanımlamalarının ardından en yaygın kullanılan iki model ile (Şelale ve Çevik) bunların kullanımlarına örnek oluşturacak birkaç yaşanmış tecrübenin karşılaştırılması yapılacaktır. Bu çalışmaya somutluk kazandıran bu bölümlerin ardından gerçek hayatta, yazılım piyasasında karşımıza çıkan benzer bir çalışma, uyarlama haline getirilecek ve bugünün piyasasında bahsedilen bu geçişin zorlukları uygulama ile ele alınacaktır (Massey ve Satao 2012).

Sonuç olarak bu çalışma ile ortaya konulması istenilen bulgu, bugünün yerel yazılım piyasasında DevOps geçiş evresini tamamlamak isteyen büyük ölçekli uygulamalar kullanmakta olan kuruluşların bu geçiş evresinde karşılaştıkları zorlukları ve tecrübeleri uygulamalar ile örneklendirerek ortaya çıkarmaktır. Bu araştırma, büyük ölçekli operasyonlara sahip olan kurum ve kuruluşların DevOps proje yönetim disiplinine geçerken yaşadığı zorluklar nelerdir sorusunu incelemektedir. Bunun yanında bu geçişin gerekliliği, geçişin nasıl bir verimlilik sağladığı sorusu ile ayrıca alt başlıklarda ve uygulamalarda incelenecektir.

BÖLÜM 2. KAYNAK ARAŞTIRMASI

Teknolojinin yazılım kolundaki son yıllarda talebin neden olduğu değişim döngüsünde çeşitli kaynaklar mevcuttur. Ancak döngü ve dönüşümün nereye veya hangi metodolojiye yapıldığına bağlı olarak kaynakların sınırlılığı artmaktadır. Yaklaşık 50 yıllık tarihe sahip olan eski disiplin anlayışı hakkında kitaplar bulunabilirken, son 10 yıldır piyasada kullanılan yeni metodolojilerin tanımlanması dışında yazılmış kaynaklar ağırlıklı olarak son 20 yılda oluşturulmuştur.

DevOps geçişleri ve eski sistem tanımlamalarından önce tanımlanması gereken ana konu sistem yaşam döngüleri olarak adlandırabileceğimiz SDLC'dir. Konu hakkında yazdığı makalesi ile Shirley Raddack, ayrıntılı bir anlatım gerçekleştirmiştir. Özellikle odaklandığımızın dışında içerdiği farklı metodolojiler ile bu çalışmaya önemli bir kaynak oluşturmaktadır (Radack, 2009).

Dougles Hughey tarafından yazılmış "Traditional Waterfall Approach" isimli makale, Şelale (Waterfall) modelinin ayrıntılı bir anlatımına şemalar ile yer vermiştir. Özellikle geçişleri Şelaleden Çeviğe (Agile) ikileminde çizdiğimiz makalemizde Şelale modelinin doğru tanımlamasını yapmak adına önemlidir (Hughey, 2009).

Tomy Mouser ve Gary Gruver tarafından yazılmış "Leading the Transformation" isimli kitabı ayrıntıları ile DevOps ve eski sistemlerin tanımlamalarını yaparken aynı zamanda eski yapıdaki sistemlere Çevik modelinin uygulanmasını anlatmaktadır. Özellikle Çevik modelin üstünde duran kitap, farklı bir sistemden çevik modeline geçişte karşılaşılan durumları, farklı modelleri Çevik model ile karşılaştırarak

sunmaktadır. Bu bağlamda kaynak olarak artı ve eksi yanları ortaya koymak konusunda örnekler ile yardım sağlayacak bir kaynaktır (Gruver ve Mouser 2015).

Anthony Lauder ve Stuart Kent tarfından yazılmış "Legacy Systems Anti-Pattern and Pattern Oriented Migration Response" isimli makale anti-pattern kavramını anlamamızda yardımcı olmaktadır. Aynı zamanda Anti-pattern kavramının DevOps ile ilişkisini ve eski sistemler ile bağını açık bir şekilde ortaya koymaktadır (Lauder ve Kent 2000).

Anthony Lauder tarafından yazılmış "Legacy systems: Assets or Liabilities" isimli kitapta normalden farklıca yapılmış bir sistem eskimesi ve kullanışsızlığı tanımı, sistemlerde karşılaşılan ve evirilmenin temel nedenlerinden birisi olan verimlilik problemi ile açık bir şekilde anlatılmıştır. Alışılagelmemiş bu tanım ile birlikte eski sistemlerden DevOps'a neden geçilmesi gerektiği ayrıca farklı bir tanımlama ile anlatmıştır (Lauder ve Lind 2007).

Çalışmanın odaklandığı belirli tanımlamaların yapılmasının ardından sistem geçişlerinde ve her iki sistemde karşılaşılan problemler çalışma boyunca ortaya konulmuştur. Bu bağlamda, Tony Bradley tarafından kaleme alınan "Moving from legacy to DevOps is a journey that takes time" isimli makale DevOps geçişlerinde özellikle büyük ve eski yapılı kuruluşların yaşadığı zorluklar tanımlanmıştır. Tüm ayrıntıları ile zorlukların her birisinde özellikle değinilmemesine rağmen Bradley, neden büyük firmaların daha fazla zorluk çektiğini ve bu sürecin onlar için neden daha uzun sürdüğünü ortaya koymuştur (Bradley, 2015).

IBM tarafından yayınlanmış bir diğer çalışmada ise Michael Edler, Jeff Crume, Timothy Hahn, Steven Pogue, ve Sanjeev Sharma alanında tecrübeli isimler DevOps metodolojisinin getirdiği sistem risklerini içeren bir çalışma ortaya koymuşlardır. Özellikle yapılan karşılaştırmalarda kullanılabilecek olan bu çalışma, henüz eksiklikleri veya hataları tam olarak tanımlanmamış yeni modellerin kötü yanlarını göstermesi adına çalışmaya katkılar sunmuştur (Edler ve ark. 2014).

Gary Gruver, Mike Young ve Pat Fulghum tarafından kaleme alınmış kaynak, bize Çevik modele geçişlerde karşılaşılan zorlukları örnekler ile verirken aynı zamanda yalnızca büyük ve eski yapılı sistemlerin Çevik model kullanımı ile DevOps'a geçişinde konuya odaklanmış olması ile kaynak oluşturmaktadır (Gruver ve ark. 2012).

Jason Hand tarafından yazılmış "Culture War: Struggle to Adopt DevOps" isimli bir diğer makalede DevOps geçişinde karşılaşılan en önemli problemlerden birisi olan kültür çatışmasına değinilmiştir. Bu konudaki sorunları ortaya koyan makalede aynı zamanda çeşitli çözümlere de yer verilmiştir (Hand, 2013).

Todd Waits tarafından kaleme alınan "Three Challanges to documentation for DevOps Teams" isimli makale, DevOps'a geçişlerde karşılaşılan sorunlardan birisi olan belgelendirme sorununa dikkat çekmektedir. Eski sistemlerin dönüşümünde karşılaşılan bu sorunu ortaya koyan bir kaynak olarak önemlidir (Waits, 2015).

Daha önce farklı kaynaklardan DevOps süreçlerinin getirdiği güvenlik açıklarından bahsetmiştik. Alyssa Robinson bahsedilen bu güvenlik açıklarından projelerin nasıl korunabileceğini ve DevOps'un işleyişinin temel paritesi olan otomatikleştirmenin bu konuda neler yapabildiğini "Implementing the Critical Controls in a DevOps Environment" isimli makalesinde ayrıntılı olarak anlatmıştır (Robinson, 2015).

DevOps geçişlerinin sebep ve sonuçlarını iyi ve kötü yanları ile tartışırken başlangıç için sormamız gereken önemli sorulardan birisi de bu sistemin veya proje metodolojisinin başarılı olup olmayacağıdır. Köklü ve eski bir sistem olmamasından ötürü bu konuda kesin tanımlamalar ve eksiksiz anlatımlar yoktur ancak geçmişte yaşanmış döngü ve değişimlerin tecrübeleri bu aşamada en güzel kaynak olmaktadır. Asami Novak tarafından ele alınmış ve DevOps geçişlerini başarı ile tamamlamış

şirketlerin ortak özelliklerine odaklanan yazı, bu soruyu cevaplamada yardımcı olmaktadır (Novak, 2014).

DevOps geçişlerinin, çalışma boyunca bahsedileceği ve örnekler ile tanımlanacağı üzere kolay olan bir süreç değildir. Süreci zor yapan unsurlardan birisi, karşılaşılabilecek sorunların bilinmezliğidir. Özellikle eski yapılı sistemlerde bu konunun işlenmesi bağlamında kuruluş yöneticilerin kuşkuları vardır. Ancak bu kuşkuları aşmak adına başarılı geçişleri yapan kuruluşların takip ettiği yollar ve karşılaştığı problemleri çözüm yolları Justin Nemmers tarafından kaleme alınmıştır (Nemmers, 2015).

Martin Fowler tarafından kaleme alınan “refactoring” isimli kitap, bir iş metodolojisini küçük parçalara ayırarak yönetmenin yararlarına değinirken aynı zamanda bunu DevOps metodolojisi ile birleştirerek bize eski ve büyük sistemlere sahip olan kuruluşların sistem geçişlerinde kullanabileceğimiz bir yol sunmaktadır (Fowler, 2018).

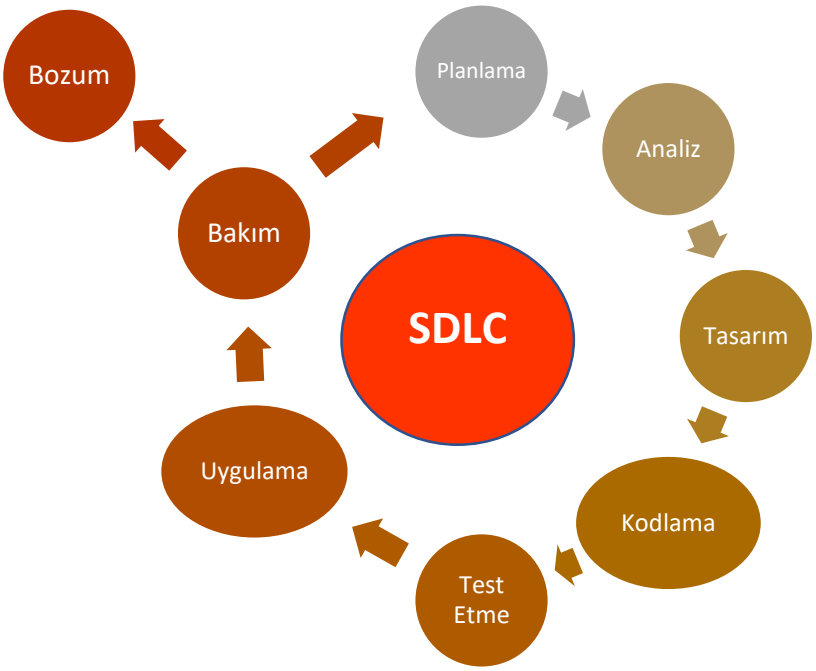
BÖLÜM 3. MODELLER

3.1. Sistem Geliştirme Yaşam Döngüsü (SDLC)

Eski ve karmaşık yapıdaki sistemler çalışmanın başlangıcında belirtildiği gibi belirli sebeplerle doğan talep karşılığında, istenilen verimliliği sağlayamaması nedeni ile birtakım evrimleşme süreçlerine ihtiyaç duymuştur. Özellikle günümüz teknolojisinin ana üretim faktörlerinden birisi olan yazılım konusunda yaşanan bu verimsizlik, kuruluşları ve proje yöneticilerini farklı disiplinleri izlemeye yönlendirmiştir. Bu durumda oluşan metodoloji geçişi ihtiyacı ve geçiş aşamasında oluşan zorlukları konu alan çalışmanın başında, ortaya çıkan ihtiyacın nedenlerini daha kapsamlı anlamak için sistemlerin temel öğelerinden birisi olan yaşam döngüsünün tanımlamasını yapmak gerekir. Genel olarak tanımlanmış şekli ile herhangi bir uygulamanın yaşam döngüsü, bize gerçekleşmekte olan evrimin tam olarak hangi aşamalarda çalıştığını anlamamızda yardımcı olacaktır.

Genel olarak Sistem geliştirme yaşam döngüsü (SDLC) şu şekilde tanımlanmaktadır; "Sistem geliştirme yaşam döngüsü, bilgi sistemlerini çok adımlı bir süreç aracılığı ile geliştirme uygulama, test etme ve bitirme adımlarını kapsayan genel bir sürecidir". Günümüzde çeşitli SDLC modelleri olmakla birlikte, her bir farklı modelinde tanımlanmış belirli adımları vardır. Yalnızca bu adımların sınırları modelden modele göre değişiklik göstermektedir (Radack, 2009).

Aşağıdaki şema, genel anlamda SDLC'nin tanımlanmasında görsel bir unsur olarak sistemlerin yaşam döngüsünü anlatmaktadır.



Şekil 3.1. Sistem Geliştirme Yaşam Döngüsünün Evreleri (Radack, 2009)

Şekil 3.1.'de görülebileceği gibi SDLC olarak nitelendirilen, bir projenin yaşam döngüsü, birkaç farklı adıma sahiptir. Burada gösterilen başlangıç aşaması, herhangi bir kuruluşun sistemsel olarak belirlediği ihtiyacını bilişim ekibine sunması olarak tanımlanır. Bu aşamada taleplere göre ihtiyaçlar belirlenir ve şirket koşullarına göre ihtiyaçlar şekillendirilerek proje halini alır. Bu aşamadan sonraki aşamalar test aşamasına kadar geliştirme aşaması olarak adlandırılır. Bu kısımda belirtilen ihtiyaçlar dâhilinde yazılım oluşturulur. Öncelikle projenin süreçleri ve planlamaları, yazılımın yapımı ile birlikte analiz edilir. Ardından çözümlerin bağdaşırım seçenekleri belirli yollar ile tasarlanır ve mimarisi oluşturulur. Sonrasında yazılımın çalışması adına yapılmış tasarıma göre kodlar yazılır ve hataların tespiti için test aşaması başlar. Burada önemli olan kısımlardan birisi, sistemin güvenliği adına atılması gereken adımların dikkatli bir şekilde belirlenmesidir. Aksi takdirde risk unsurunda fark edilmeyen bir artış ile bir sonraki aşamaya geçilir ki bu projenin

uygulama kısmında, kuruluş adına güvenlik risklerini de beraberinde getirir. Test aşamasından önce belirlenen güvenlik önlemleri, testler ile birlikte test edilerek proje yürürlüğe geçmek üzere yönetici onayına bırakılır. Uygulama aşamasında ise sistem artık müşterilere açık haldedir ve kullanım halinde olan sisteme olası durumlarda ihtiyaç duyulan bakımlar yapılır ve eksiklikler giderilir. Bakım kısmının sonunda sistemin durumuna göre izlenebilecek iki yol mevcuttur. İlk olarak sistem küçük ihtiyaçlara sahip ise gerekli bakımların ardından yeniden başlangıç aşamasına döndürülerek eksiklikleri giderilir ve operasyona alınır. İhtiyaçların ve eksikliklerin fazlalığı, operasyona kadar olan süreçteki adımların daha uzun ve maliyetli olmasına neden olmaktadır. İkinci ihtimal ise bozum aşamasıdır. Bozum aşaması bir sistemin yaşam döngüsünün son aşaması olarak tanımlanabilir. Bu aşamada adından da anlaşılacağı üzere mevcut bir sistemdeki donanım, yazılım ve veri içerikleri bir plan dâhilinde yeni oluşturulacak farklı bir sisteme aktarılır. Böylece eski sistem kullanım açısından emekli edilmiş ve kullanımdan kaldırılıp yerine tamamen yeni bir sistem getirilmiş olunur (Radack, 2009).

Ancak genel olarak burada bahsedildiği gibi sistemlerin kesin bir son aşaması yoktur. Genellikle uygulamalar veya sistemlerde ihtiyaçlar projeler dâhilinde geliştirme ve ekleme yolu ile giderilir. Yalnızca bazı nadir görülen karmaşık ve parçalanarak bakıma sokulması mümkün olmayan sistemlerde yeniden yapılandırmalardan söz edebiliriz. Bunun dışında genel anlamda takip edilen yöntem, sistemlerin tamamen silinip yerine yeni sistemlerin gelmesi değil, ihtiyaçlara veya değişen teknolojilere göre kendi içlerinde geliştirilmeleridir. Bu nedenle böyle durumlarda bozum aşaması olarak adlandırdığımız aşama atlanır ve kullanımdaki aynı sistem güncellemeler ile birlikte uygulama aşamasına döner (Radack, 2005).

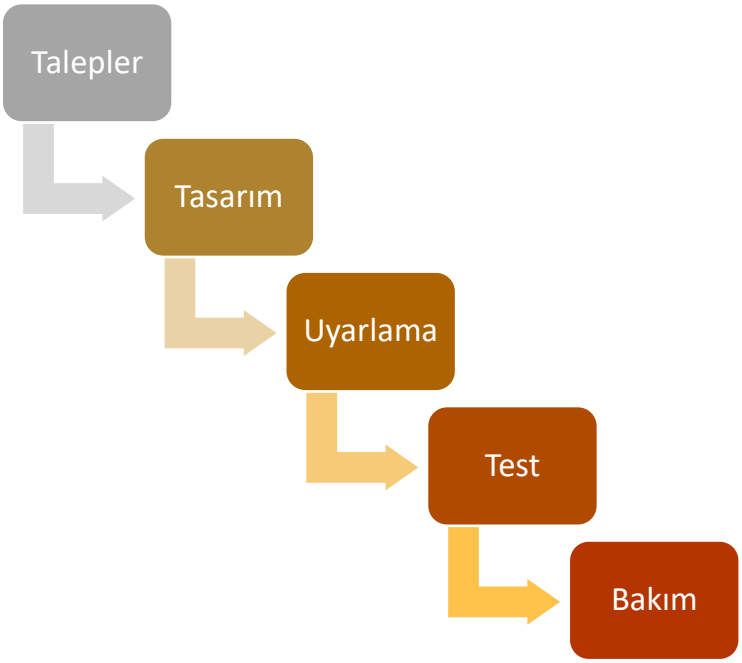
Önceki paragraflarda bahsettiğimiz üzere, farklı çeşitlerde SDLC modelleri ile karşılaşmak mümkündür. Bunlar yazılımcıların veya şirketlerin kültürüne ve ihtiyaçlarına göre değişiklik gösterebilirler. Bunlardan en yaygın olarak kullanılan ikisi Şelale ve Çevik olarak adlandırdığımız modellerdir. Şelale modeli, istenen projenin tüm gereksinimlerinin anlaşıldığı ve projede eksiksiz şekilde bahsedildiği

veya türleri veya kurumlar adına kullanışlı bir modeldir. Ancak bu durum her proje için mümkün olmamakla birlikte, genel anlamda ulaşılması zor bir tanımlamadır. Diğer bir yandan Çevik model ise tanımlamaları en başta kesin olarak yapılamayan ve olası problemleri kesin olarak öngörülemeyen veya öngörülmesi maliyetli olan projeler için kullanılmaktadır. Bir nevi uyarlanabilir SDLC olarak da adlandırılmaktadır (Amlani, 2012).

3.2. Şelale (Waterfall) Modeli

Doğrusal şekilde planlamalardan oluşan Şelale modeli ve yinelemeli kontrol mekanizmasına sahip Çevik modelinin kökenleri 1930'larda Bell Labaratuvarlarında kalite uzmanı olarak çalışan Walter Shewhard'ın "planla – yap – çalış – harekete geçir" disiplini olarak bilinen modeline dayanmaktadır. Şelale modeli olarak adlandırılan bu model ilk olarak 1956'da Herbert Benington tarafından basamaklı bir şelale gibi tanımlanan bir yazılım geliştirme modeli olarak ortaya çıkartılmıştır. 1970 yılında ise Winston W. Royce, bu modeli ele alarak daha kapsamlı halde incelemiş ve günümüzdeki büyük ölçekli projelerin planlanmasında kullanılır haline getirmiştir. Yapılan son eklemeler ile birlikte Şelale modeli, bir önceki adımın bitirilmesi şartı ile bir sonraki adıma geçişin sağlandığı bir model olarak günümüzde kullanılmaktadır (Amlani, 2012; Font, 2016).

Bahsedilen bu modelin gelişim aşamaları Şekil 3.2. de gösterilmiştir. Her bir adım uygulama projesinde farklı bir kategoriyi temsil etmektedir. Bir sonraki adımda bir önceki adıma dair bir işlem yapmaktan kaçınılmakta ve her bir adımda o adımın konusuna odaklanılmaktadır.



Şekil 3.2. Şelale Modeli düzlemsel aşamaları

Talepler aşaması olarak adlandırılan ilk aşamada geliştirme ekibi, ilgili kurumun taleplerine göre projenin en başında gereksinimlerini analiz eder ve uygulamanın, işletme ve teknik yönlerini ayrıntılı olarak içeren bir rapor haline getirir. Sonraki tasarım aşamasında geliştirme ekibi veri tabanı tasarımı, donanım gereksinimleri, yazılım gereksinimleri ve iş akışını belirten bir uygulama mimarisi oluşturur. Yine bu aşamada yazılım ekibi uygulamanın işlevselliğini projeye bağlı kalarak geliştirir ve uyarlama aşaması için uygulamayı hazırlarlar (Hughey, 2009).

Şelale modelinde, operasyon ekibi uyarlama/uygulama aşamasından sonra gelen yazılım geliştirme aşamasında projeye öncülük eder. Kullanımı ile ilgili kontrolleri gerçekleştirmelerinin ardından projeyi test aşamasına gönderir. Test aşamasında ise yazılım ekibinin yanında operasyon ekipleri ayrıca tüm uygulamanın işlevselliğini test eder. Operasyon ekibinin testleri tamamlamasının akabinde uygulama, müşteri karşısına çıkmak için onay almak adına yönetime sunulur. Bundan sonra da uygulama, operasyon ekipleri tarafından yapılacak bir sonraki hata kontrolü ve kod optimizasyonu sürecine kadar bakım halinde çalışır. Şelale modeline göre

kurgulanmış bir projede, sistem geliştirme yaşam döngüsü sırasında uygulamanın veya projenin tasarımı hakkında bir problem oluşursa, projenin ikinci veya üçüncü aşamaya dönmesi ve tüm bu süreci yeniden başlatması gerekir. Bu nedenle, doğrulama veya test aşamasında fark edilen bir eksiği geri almak ve en baştan değiştirmek çok zorlu bir süreçtir (Hughey, 2009).

Şelale modeli, büyük veya projelerini çok detaylı tasarlayabilecek yetiye sahip kuruluşların kullanımı adına faydalı olabilmektedir. Bu faydalardan bazıları, model kapsamında başlangıçta verilen proje gereksinimlerinin çok açık bir şekilde belirlenmesi, kodlama sırasında gereksinimlerin değişmemesi ve projelerin adımlarının kolayca kontrol edilebilmesidir. Bu modelde uygulama geliştirme süreci boyunca fazla esneklik payı olmazken, proje üzerinde yapılacak ani karar değişikliklerine kesinlikle yer yoktur. Gelişimin her aşaması, kesin talimatlar ve tarihler ile belirlenmiştir. Proje adına yapılacaklar ve gelecek planlamaları, projenin en başta açık bir şekilde tanımlanmasından dolayı gayet kolay ve işlevseldir. Hem süre hem de sonuçlar iyi belgelenmiştir. Ancak ne yazık ki eksik kalan kısımların beklentileri karşılamaması ve oluşturulan projelerin amacına uymaması durumunda bahsedilen tüm avantajlar dezavantaja dönüşebilmektedir (Massey ve Satao, 2012).

Büyük organizasyonlara ve karmaşık uygulama yapılarına sahip şirketlerin yönetimleri Şelale modelini, sahip olduğu disiplin sonucunda tercih etmektedirler. Sistemsel olarak oluşturulan projenin ilerlemesini basamaklar halinde izlemeyi kolaylaştıran adımlar, yöneticiler tarafından genellikle çekici bulunmaktadır. Bu model aynı zamanda sahip olduğu katı disiplini nedeniyle yönetimin örgütlenmesini kolaylaştırır çünkü her aşamada o aşama için tanımlanmış faaliyetler ve teslim edilebilir sonuçlar belirlidir (Dale, 2016).

Şelale modeli, diğer modeller ile karşılaştırıldığında daha öngörülebilir ve net bir teslim tarihine sahiptir. Daha önce bahsedildiği üzere şelale modelindeki başlangıç noktası, ihtiyaçların tam olarak belirlenerek projenin belirlenen ihtiyaçlara göre

oluşturulmasıdır. Bu durumda eksik belirlenen her ihtiyaç aynı zamanda projenin sonunda eksik veya hata ile karşılaşılma riskini ortaya çıkarmaktadır. Bu nedenle mümkün olduğu kadar, riski ortadan kaldırmak için projeler tamamen araştırılmakta ve her adım, ihtiyaçlara göre belgelenmektedir. Önceden gerçekleştirilen araştırmalar ve ortaya konan ihtiyaçlar neticesinde proje teslim tarihi daha öngörülebilir bir hal almaktadır (Massey ve Satao, 2012).

Şelale modeli, farklı proje modelleri ile kıyaslandığında (Projenin yapısına bağlı olarak) bazı durumlarda gerek zaman gerek ise maliyet bakımından tasarruf sağlayabilmektedir. Bu durum projeden projeye veya talebe göre değişiklik gösteriyor olsa da genel anlamda doğru bir şekilde organize edilmiş projelerde bu durumu görmek mümkündür. Değinilen düzlemsel planlama stili, her adımın tamamlanmış bir bitiş noktası ile birlikte bunun yanı sıra her adımın kendine ait belirli görevleri olduğu bir disipline sahiptir. Bu nedenle Şelale modeli, gereksinimleri detaylı ve düzenli bir şekilde tanımlanmış, teknolojisi ve ihtiyaçları iyi bir şekilde anlaşılmış, farklı tanımlamaları olmayan gereksinimlere sahip ve destek için gerekli uzmanlığa sahip personeli olan kurumların projeleri için güzel bir çalışma sergileyebilir. Ayrıca projenin başlangıcında belirlenen bütçenin aşılması oldukça az ihtimale sahip bir durumdur. Bu durumdan dolayı bahsedilen maliyet ve zaman konusundaki tasarruf meydana gelebilmektedir. Yani belirli bir bitiş tarihi ve planlanmış maliyet analizi bu modeli sınırları belirli bir yapıya sokmaktadır. Ancak genel anlamda proje yönetim modellerinde durum anlatıldığı gibi gelişmemektedir. Gözden kaçan herhangi bir eksik veya hatanın projenin tamamını etkileyeceğinden dolayı taşıdığı risk göz ardı edilmemelidir (Hughey, 2009).

Şelale modelinin her bir aşamasının bir sonraki aşamadan önce tamamlanması gerektiğinden, müşterilerin üretimden önce talep ettikleri uygulama hakkında geri bildirimde bulunmaları veya projeyi görmeleri olanaksızdır. Müşteri tarafından herhangi bir kontrol talebi veya değişiklik talebi geliştirme sürecinde gelse bile, geri dönüş hem zaman hem de maddi bakımdan maliyetli olduğundan dolayı, istenen talebi gerçekleştirmek veya kabul etmek, diğer modellere kıyasla daha az

mümkündür. Buna bağlı olarak teslim sonrasında projede belirtilmemiş ancak müşterinin isteğine veya iş yapısına uygun olmayan durumlar için projeyi geri döndürmek ve tekrar yazımını sağlamak projenin görünmeyen maliyeti olarak tanımlanmaktadır (Hughey, 2009).

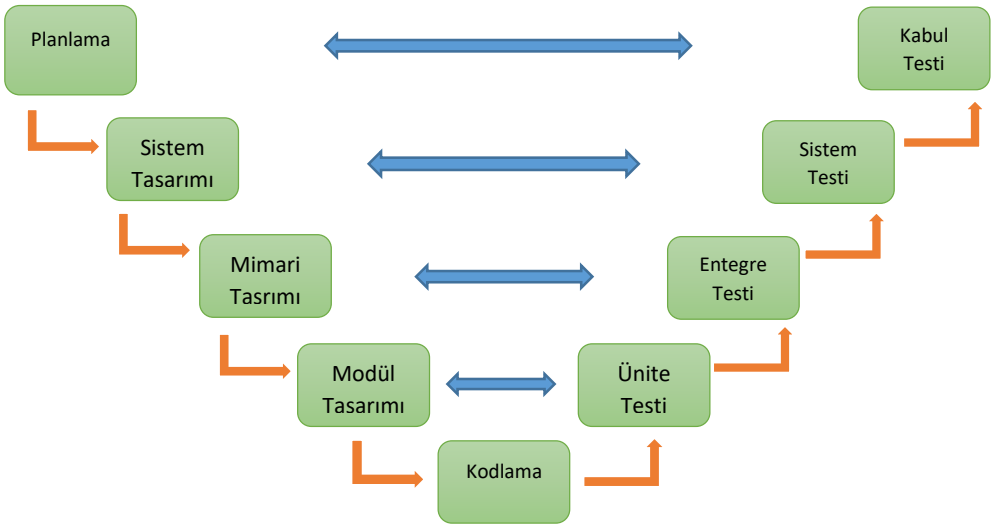
Daha önce bahsedildiği üzere, Şelale modelinin organizasyon yapısı mükemmel olmamakla birlikte avantaj olarak gözüken birtakım dezavantajlar içermektedir. Projenin en başında belirlenen gereksinimler ve bu gereksinimlerin gelişim aşamasında yeniden tartışılmaması, model için başlıca sorunu teşkil etmektedir. Proje döngüsünün başında yer alan ve belirlenen varsayımların birçoğu gelişme sırasında kullanılmaz hale gelebilmekte, başlangıçta küçük veya fark edilmeyen problemler daha sonrasında büyük sorunlara sebep oluşturabilmektedir. Yazılımın üretiminin uzun zaman alması, kullanıcının uygulamayı en son aşamada görebilmesi ve bu nedenle yazılımın kullanıcı ihtiyaçlarına cevap verememesi, modelin katlılığının gerekli esnek değişikliklere izin vermemesi ve bu değişim koşullarının gözükmemesi gibi sebepler, modelin en temel problemlerini oluşturmaktadır. Özellikle projenin yaşam geliştirme ve yaşam döngüsünün uzunluğu, yeni bir evrimin ihtiyaç duyulmasına varan taleplere ve sonuçlara sebebiyet vermektedir (Gruver ve Mouser, 2015).

Bunun yanında Şelale modeli esnek olmayan yapısı itibariyle her proje ile iyi çalışmayabilir. Buna en güzel örnek, güncellemesi çok olan veya farklı alanlarda iş kollarına sahip kuruluşlardır. Bununla birlikte aynı sebeple uzun süren yazılım geliştirme sürecinde oluşacak herhangi bir sektöre bağlı değişken, modelin çalışmasını olanaksız veya maliyetli hale getirebilmektedir. Aynı zamanda, projeyi her aşamasında kontrol etmek isteyen veya bunun gerekli olduğu iş kolları adına bu model uygun değildir (Hughey, 2009).

3.3. V-Şekilli Model

Bu model, bir önceki süreçte tanımladığımız şelale modeli ile hemen hemen aynı mantığı taşımaktadır. Şelale modelinde olduğu gibi katı bir disipline sahip olan bu modelde de bir sonraki aşamanın başlangıcı, önceki aşamanın bitişi ile mümkün olmaktadır. Ancak Şelale modelinden farklı olarak proje basamakları lineer olarak aşağıya doğru gitmek yerine, her modelin eşleniği olarak bir test evresini barındırmak sureti ile V şeklini çizen bir şemaya sahiptir.

Model, ilk önce Şelale modelinde olduğu planlama evresi ile başlar ve Şelale modelindeki adımları kodlama aşamasına kadar aynı şekilde takip eder. Ancak kodlama aşamasından sonra yukarıya doğru yönelen lineer gelişim evresi, çoğunlukla kontrol ve test süreçlerini kapsayan ve test süreçleri sonucunda uygulamaya geçen bir süreci temsil eder. Her test süreci, aslında gelişim evresinde karşılık bulduğu adımın bir testi niteliğindedir. Şekil 3.3.'de görüldüğü gibi V- Şekli model toplamda dokuz ana basamaktan oluşmaktadır (Mircea ve ark. 2013).



Şekil 3.3. V-Şekli Proje Modeli (Mircea ve ark., 2013).

Normal olarak Şelale modelinde karşılaştığımız tasarım aşamaları birkaç farklı parçaya bölünmüş ve her bir planlama aşamasında o aşama ile ilgili test oluşturulmuştur. Planlama aşamalarının bitmesinin ardından gelen kodlama aşamasından sonra her aşama için önceden belirlenmiş şekilde ilgili alandaki testler uygulandıktan sonra proje hayata geçirilir. Bu nedenle esneklik yapısı en az olan model olarak bilinir.

Karşılaştırmalı olarak bakıldığında, V-Şekil Proje modeli Şelale modeline çok benzemektedir. İkisinin de disiplininde bir önceki adım bitmeden bir sonraki aşamaya geçilmez. Ancak Şelale modelinden farklı olarak V-Şekil modelde testler, tasarımların oluşturulduğu aşamada önceden belirlenirler. Bu da proje geliştirme modeli olarak V-şekil modeli diğer modellere göre daha güvenilir hale getirir. Ancak bu güvenilirlik, proje oluşumunda yapılan araştırmaların derinliğine ve kalitesine bağlı kalır. Önceden öngörülemeyen risklerin sonraki aşamalarda probleme dönüşümü, bu proje modeli için en temel risk unsurunu oluşturur. Bu risk unsurunun nedeni olan kısıtlı esneklik yapısı, Çevik model ile karşılaştırıldığında bu model için en temel zayıf noktadır. Bu nedenle iyi planlanmış büyük sistemler veya planlanması çok karmaşık olmayan küçük sistemler için tercih edilebilen bir yöntem olmasına rağmen, her zaman öngörülemeyen risk unsurunu barındırır. Bunun yanında Çevik modeli ile benzer olan bir yanı mevcuttur. Bazı durumlarda Çevik modelin testleri, tasarımın oluşturulma aşamasında belirlenebilir. Bu gibi Çevik model projeleri, testlerin oluşturulma zamanları ile V-Şekil modele benzerlik göstermektedirler (Radack, 2009).

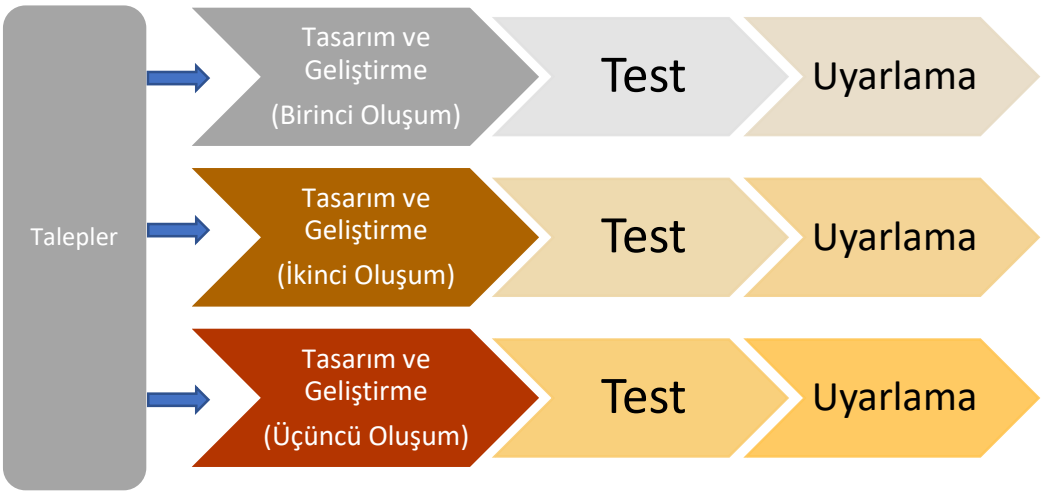
3.4. Artırımlı ve Yinelemeli Model (Incremental and Iterative Model)

Proje yaşam döngülerinde kullanılan bir diğer model Yinelemeli model (Iterative Model) olarak bilinir. Bunun yanında ayrı olarak tanımlanan ancak birbirleri için tamamlayıcılık özelliği gösteren bir diğer model ise Artırımlı (Incremental) modeldir. Her iki model de kullanımı ve işleyişi bakımından ayrı ayrı bakıldığında

Çevik model ile bazı benzerliklere sahiptirler ve birleştiklerinde Çevik model manifestosunun tanım olarak önemli bir kısmını oluştururlar. Bu nedenle Çevik model tanımlamalarında bu iki model, Çevik modelin esnekliğini sağlayan iki temel unsur olarak görülmektedir.

Öncelikle Yinelemeli model, Şelale modelinde görülen esnek olmayan proje disiplini yüzünden oluşan bir ihtiyaç sonucu ortaya çıkmış modeller arasındadır. Bu modelde projenin müşteriye gösterilmesi ve dağıtımının gerçekleşmesi için projenin tamamlanması gerekmez. Bunun yerine proje küçük parçalara bölünür ve bu parçalar sonuca ulaştırılmak üzere geliştirilmeye başlanır. Projenin tanımı daha çok projenin geliştirilmesi aşamasında gerçekleşmektedir. Bu model bir nevi Şelale ve Çevik modellerinin bir karışımı olarak tabir edilebilir.

Yinelemeli modelde projenin gerekli planlaması taleplerin toplanmasının ardından yapılarak, proje farklı parçalara bölünür. Bütün projenin bitiminden sonra müşteri görüşmesi gerçekleşmesi yerine parçalara ayrılmış bölümlerden birisinin bitmesi ile biten kısım ile ilgili görüşme gerçekleşir. Bu nedenle projenin gelişim döngüleri Şelale modeline göre daha kısadır. Buraya kadar Çevik modele benzese bile bölünmüş her bir parçanın tasarımı ve geliştirilmesinde Şelale modelini izler. Diğer bir söylem ile bütün bir projeyi geri dönülmez bir disiplin içerisinde geliştirmek yerine, parçalara ayırarak Şelale modelini parçalarda uygulamayı tercih eder. Böylece Şelale modelindeki öngörülemeyen hatalar riskini düşürmüş olur. Ancak bu yönü ile de esnekliğini kaybederek Çevik modelden ayrılır.



Şekil 3.4. Yinelemeli Model (Mishra ve Dubey, 2013).

Şekil 3.4.'te görüldüğü üzere örnek olarak talepleri toplanmış bir proje üç kısma bölünerek kurgulanır. Bu ayrımdan sonra ikinci aşama olan geliştirme aşamasına geçilir. Geliştirme aşamasında oluşum olarak ayrılan her kısım ayrı ayrı geliştirilerek testlere tabi tutulur. Testlerden sonra gerekli görülmesi halinde eksiklikler, uyarlama ve bakımlar ile sağlanır ve müşteriye teslim gerçekleşir. Yapı bakımından esneklik ve ihtiyaçlara sonradan cevap verebilme imkânına sahip olan bu proje modelinde yaşanan sorun, Şelale modelinde olduğu gibi, ayrılmış kısımların tesliminde oluşacak problemlerde geri dönüş zorluğudur. Ancak bu zorluk bütün projenin yalnızca bir kısmına hitap ettiği için getireceği masraf klasik şelale modelinden çok daha az olacaktır. Bu özelliği ile riski ve maliyeti düşürebilse de, projeler için Çevik modelde olduğu kadar geniş bir esneklik sağlamamaktadır. Sahip olduğu bu özelliği nedeni ile modelde eksik olan kısım daha sonra Artırımlı model ile giderilmeye çalışılmıştır (Radack ve ark., 2009).

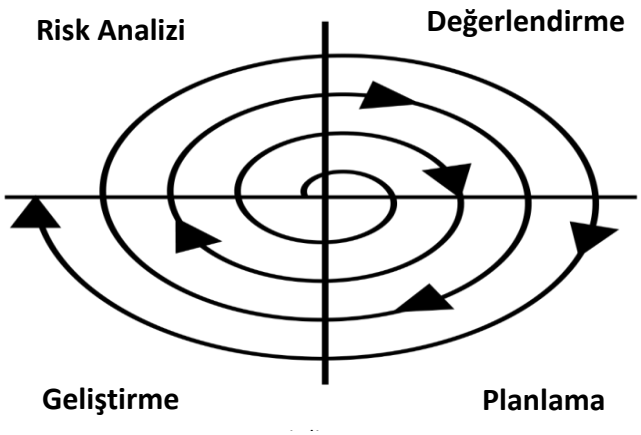
Artırımlı modelde, yinelemeli modelden farklı olarak proje prototipler üzerinden ilerler. Öncelikle yinelemeli modeldeki gibi bir proje planlama aşaması ile başlayan bu model, en basit anlamda ihtiyaçlara karşılık verebilen ve belirli testlerden geçebilecek örnek uygulamanın oluşması ile gelişme evresini tamamlar. Daha sonra

müşteri kuruluş ile yapılan ikili görüşmeler sırasında belirlenen eksik noktalar bir sonraki gelişme evresinin planını oluşturur. Bu döngü proje tamamlanana kadar devam etmektedir. Esneklik yapıları ile Çevik modele benzeyen bu iki model, ancak birleşerek Çevik modelin temel genel prensiplerinden birisini oluşturabilirler. Çevik modelden farklı olan tarafları, ayrıntılı bir şekilde Çevik modelin tanımlaması sırasında anlatılacaktır (Radack, 2009).

3.5. Spiral Model

Artırımlı proje modeline benzeyen bu model, genellikle en az 6 aylık en fazla da 2 yıllık sürekli döngü süreçlerinden oluşur. Her bir döngü sürecinin bitimi ilgili döngü sürecinin kapsamındaki konunun üstünde yapılan çalışmanın bitimi ile eşleşmektedir. Ayrıca her döngü sürecinin sonunda müşteri ile tamamlanan döngünün aşamaları tartışılarak eksiklikler gözden geçirilir (Raddack, 2009).

Spiral model, sağladığı kısa müşteri görüşmeleri ve bunun ile birlikte ortaya çıkan takım çalışmaları ile Çevik modele en çok benzeyen modeldir. Bu model, aynı zamanda artırımlı ve yinelemeli modellerin özelliklerinin ikisini de bir arada bulundurmaktadır. Başlangıç noktası veya proje planlama evresi, spiral görünümündeki şemanın ortasından başlamaktadır ve döngüler saat yönünde ilerleyerek farklı konulara odaklı aşamalardan geçmektedir. Dört ana aşamanın da tamamlandığı her bir döngünün sonunda ya bir prototip ya da temel özelliklere sahip ve dağıtıma hazır bir uygulama müşteriye sunulmaktadır.



Şekil 3.5. Spiral Model (Mishra ve Dubey, 2013).

Şekil 3.5.'te belirtilen modelin olumlu yanlarında risk analizinin sıkça yapılması, aynı zamanda artırımlı model ile olan farkını da ortadan kaldırmaktadır. Çevik modele kıyas ile neredeyse aynı esneklik yapısına sahip bu modelin olumsuz tarafı, belirli bir bitiş tarihinin olmamasıdır. Döngüler ve geliştirme aşaması proje tamamlanıncaya kadar devam eder. Sahip olduğu disiplin ile önceki esnek modellere benzer şekilde küçük ve karmaşık olmayan projelerin sağlıklı bir şekilde tamamlanması adına uygun bir modeldir. Ancak belirsiz teslim süresi nedeni ile karmaşık yapıdaki sistemlere uyarlanmasında bazı sorunlar yaşanabilir. Bu durum maliyetli bir iş sürecini gerektirebilir. Bunun dışında her alan eşit öneme sahip değildir. Örneğin risk alanında yapılacak testlerin önemi ile plan aşamasının önemi aynı değildir. Bu nedenle aynı zaman dilimini kapsamamaları gerekmektedir (Massey ve Satao, 2009).

3.6. Çevik (Agile) Model

Çevik model, Şelale modelinin tam aksine, isminden de anlaşılacağı üzere daha esnek değişim yapısına sahip ve bu esnek yapısı sayesinde müşteri isteklerine daha hızlı cevap verebilen, yinelemeli ve süreç boyunca gelişimi olan bir yazılım geliştirme modelidir. Bütün projeyi teslime kadar kademelere bölerek yazılım

gelişim yönetimini oluşturmak yerine, modelleri küçük parçalara ayırarak parçalar arasında müşteriye projeyi gösterme özelliğine sahiptir. Çevik modelin esnek proje oluşturma yapısı Şekil 3.6.'da gösterilmiştir. Her yineleme; planlama, gereksinim analizi, tasarım, kodlama, birim testi ve kabul testi gibi çeşitli alanlarda aynı anda çalışan çapraz fonksiyonel ekipleri içermektedir. Kod dağıtımı, her yinelenen küçük bölümün sonunda gerçekleşmektedir (Ambler, 2003).

Kısaca bu modelin çalışmasını anlatmak gerekirse model, birbirlerine benzer şekilde süre gelen döngülerden meydana gelmektedir. Öncelikle bahsedilen bu döngülerin en başında projenin talepler doğrultusunda planlaması yapılır. Sonrasında tasarım ve kodlama aşamalarına geçilir. Ardından yapılacak olan testler ile kötü kodlamalar ortadan kısmi olarak kaldırılır ve uygulama sisteme dâhil edilir. İlk başta çıkan basitleştirilirmiş bu versiyon, daha sonra müşteri talepleri ve kuruluşun gördüğü eksiklikler doğrultusunda 2-3 aylık bir gözlem sonrası yeniden yapılandırılır ve bu süreç proje tamamlanana kadar devam eder.

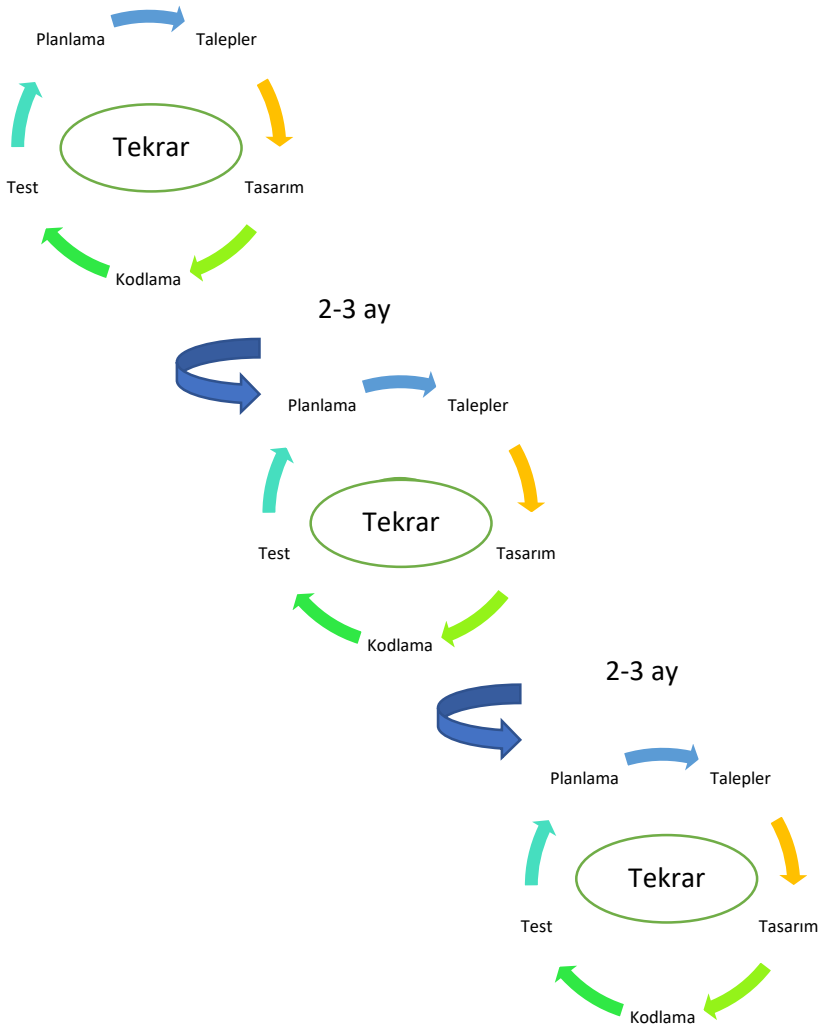
Çevik proje yaşam döngüsü modeli, sahip olduğu bu özellikleri yönünden aslında daha önce anlatılan artırımlı, yinelemeli ve özellikle de spiral modele benzemektedir. Ancak temelde yer alan bu benzerliklerin yanı sıra birtakım farklılıklara da sahiptir. Öncelikle Çevik modelin 2001'de oluşturulan manifestosuna bakıldığında göze en çok çarpan özelliği, müşteri ve operasyon birimleri ile ortaklaşa bir çalışma sonucu uygulamaların geliştirilmesidir. Bu disipline kısmen sahip diğer SDLC modellerinden farklı olarak çevik model, tek bir tanımlamadan oluşmamaktadır ve Çevik modelin yalnızca kendisine ait olan 5 farklı metodolojisi mevcuttur. Bu metodolojiler sırasıyla kristal metodoloji, Dinamik yazılım geliştirme metodolojisi (DSDM), özellik odaklı metodoloji, Scrum, ve aşırı kod odaklı metodolojilerdir.

Sözü geçen bu metodolojiler, konuyu daha karmaşık hale getiriyormuş gibi görünseler de aslında her bir metodoloji, farklı proje taleplerine hitap edebilmek ve çözüm üretebilmek adına oluşturulmuş, farklı ağırlıklara ve disiplinlere sahip çalışma

yöntemleridir. Örneğin; piyasada en çok karşılaşılan DSDM metodolojisinin temel özelliği, bir uygulama geliştirme projesini proje öncesi süreç, projenin hayata geçirildiği süreç ve proje sonrası süreç olarak 3 ana bölüme bölmek ve gelişimi bu bölümlere göre uygulamaktır.

Şekil 3.6'da bahsedilen bu bölme işlemini temsil eden tipik bir Çevik model görselleştirilmiştir. İlk olarak gerçekleşen proje öncesi süreçte, ihtiyaçlar uygulamayı talep eden kuruluş ile birlikte belirlenerek temelde oluşturulması gereken bir uygulamanın planlaması yapılır. Sonrasında tasarım ve kodlama aşaması ile uygulama belirtildiği şekilde çalışır hale getirilir ve teste tabi tutulur. Ardından müşteriye sunularak uygulamanın dağıtımı ve canlıya alım süreci gerçekleşir. Bu süreçte diğer modellerden farklı olarak, geliştirilen uygulama prototip halinde kalmaktan ziyade iyi bir şekilde geliştirilmiş ve müşteriye sunulmak üzere kullanıma hazır bir şekilde sunulur. İlk aşamanın yeterli gelmemesi halinde müşteriden alınan geri bildirimlere bağlı kalınarak talepler yerine getirilene kadar aşama tekrar edilir. Ancak bu döngü diğer modellere göre daha kısa bir zamanı kapsamaktadır. (Şekil 3.6.'da belirtilen 2-3 ay temsilidir ve projeye göre değişiklik göstermektedir).

Proje öncesi aşamanın bitişinin ardından temel ihtiyaçları karşılayacak bir şekilde işlev gören uygulama dağıtıma çıkartılır ve müşteri ile buluşturulur ve böylece projenin hayata geçirildiği sürece geçilmiş olunur. Bu süreçteki temel hedef, projenin müşteri ihtiyaçlarını karşılamasındaki işlevselliğini ölçmektir. Beklenmedik şekilde ortaya çıkan problemler projenin hayata geçtiği süreç boyunca yeni döngünün planlama aşamasını oluşturur.



Şekil 3.6. Çevik Model (Mishra ve Dubey, 2013).

Gerekli güncellemeler yardımı ile bitirilen bu sürecin sonunda proje sonrası aşamaya geçilir. Burada temel hedef uygulamanın farklı fonksiyonlar kazandırılarak geliştirilmesidir (Mircea ve ark., 2013).

Çevik modelin altında yer alan bir diğer metodoloji Kristal metodoloji olarak adlandırılır. Bu metodolojide uygulamanın yapısından ziyade uygulamada yer alacak ekiplerin dağılımı söz konusudur. Fazla karmaşık yapılara sahip kuruluşların entegresinde kullanılmaya uygun olan bu metodolojide geliştirme aşaması uygulama küçük parçalara ayrılarak ilgili takımlara devredilir ancak farklı parçalara ayrılan

ekipler birbirlerinden farklı projelerde görev alsalar dahi yaptıkları geliştirme işlemlerini birbirleri ile koordineli olarak çalışarak gerçekleştirirler (Mircea ve ark., 2013).

Özellik odaklı metodoloji, çalışma yapısı bakımında Artırımlı modele benzemektedir. Buradaki sistem, ekiplerin veya uygulamanın bir bütün olarak geliştirilmesi yerine uygulamanın fonksiyonlarına göre alt kısımlara ayrılarak geliştirilmesi disiplinini benimsemiştir. Artırımlı modelden farkı, sahip olduğu daha kısa süreli zaman döngüleridir. Bu metodoloji, kristal model gibi karmaşık yapılı işlere sahip kuruluşların uygulamalarını geliştirmeye uygundur.

Scrum metodolojisi, ismini Amerikan futbolundaki hiyerarşik dizlimden almaktadır. Bunun nedeni hem karmaşık yapılara hem de bu yapıların çözümlemelerinin ve taleplerinin net olmadığı kuruluşların, uygulama geliştirme stratejilerini sahip oldukları problemlerden yola çıkarak oluşturmalarıdır. Bu duruma bağlı olarak uygulama geliştirme aşamaları da büyükten küçüğe doğru giden ve ana problemlerin altında yer alan alt problemleri barındıran hiyerarşik bit yapıya sahip olurlar. Bu metodolojide döngüler yerine kat edilen her aşamanın günlük olarak kuruluşa bildirildiği sistemler vardır. Bildirimler sonucunda müşteri olarak uygulamayı talep eden kuruluşlar bir sonraki sprint olarak adlandırılan aşamanın ne olması gerektiğini tanımlarlar. Ayrıca her ekipte, ekibin verimli çalışmasını önleyen problemlerin çözümü ile görevli bir kişi bulunmaktadır. Gerek ekip stratejileri gerekse günlük verilen geri bildirimleri ile Çevik modelin en esnek yapılı metodolojisidir (Mircea ve ark., 2013).

Son olarak Aşırı kod odaklı metodoloji (XP), genel olarak uygulama tesliminin olmadığı ve gelişimin sürekli olduğu kuruluşlarda kullanılmaktadır. Diğer metodolojilerden farklı olarak XP metodolojisi kendi içinde dönüşüm geçirmiş bir metodolojidir. Önceden kullanılan XP metodolojisi 12 adımdan oluşan bir döngüye sahiptir. Bunlar; oyunun kurgulanması, küçük çaplı kod dağıtımları, metafor, basit

planlama, test, yeniden yapılandırma, eşleşmeli programlama, işbirlikçi sahiplik, sürekli entegrasyon, 40 saat çalışmalı hafta, yazılımlar ve kod standartlarıdır. Bu döngüdeki adımların hepsi sürekli gelişim gerektiren uygulamaların ve kuruluş sistemlerinin takip etmesi ve kontrol etmesi gereken başlıklardır. Bunun yanında sonradan geliştirilen ve günümüzde aktif olarak kullanılan yeni XP bu bölümlerden ayrı olarak farklı pratiklere sahiptir. Bu pratikler; takım bütünlüğü, bilgilendirici çalışma alanı, baskı altında çalışma, eşlemeli programlama, hikayeler, haftalık döngüler, üç aylık dönemler, 10 dakikalık uygulama geliştirme süreçleri, devamlı entegrasyon ve yenilemeli tasarımdır. Sıralanan bu pratikler, bu tarz Çevik bir metodolojinin uygulanmasında oluşturulmuş bir nevi kurallar dizilimidir. Sürekli gelişimi gerekli kılan bir çevik metodolojinin varlığı, sistem gereksinimlerindeki yenilenme ihtiyacının ne kadar dar bir süreç haline geldiğini göstermektedir. Bu gibi ihtiyaçlara eski disiplinli Şelale modeli cevap veremezken, Çevik model bu ihtiyacı karşılayabilmek adına özel bir metodoloji oluşturmuştur. Bu gibi bir metodolojide artık iki farklı (Geliştirici ve Operasyon) kuruluştan bahsetmek yerine, sürekli olması gereken gelişim evresi gereği ikisini bir çatı altında barındıran kuruluşlardan söz etmeye başlayabiliriz. Teziminiz ilerleyen sayfalarında tanımlayacağımız DevOps disiplini bu metodoloji ile tam anlamda uygunluk göstermektedir (Mircea ve ark., 2013).

BÖLÜM 4. MODELLERİN UYGULANMASI

4.1. Çevik Modelin Eski Model SDLC'ler ile Karşılaştırılması

Genel anlamda Çevik modelin tanımını ve sahip olduğu farklı disiplinler ile birlikte metodolojilerini anlattıktan sonra, çalışmanın bu bölümünde bu modelin hangi yönleri ile diğer modellerden ayrıştığını ve bu ayrışma sonucu kullanım anlamında günümüzde bir popülerlik yakaladığını bu bölümde belirteceğiz

Eski yöntemler kullanarak proje geliştirme sistemlerine sahip ve bu sistemlerin kültürel yapısı ile planlanmış uzun vadeli kod teslimatları, tüm yazılım geliştirme projelerinin yaklaşık yarısını oluşturmaktadır. Proje için gereken çözümler, yaşanan problemlerin müşteriler tarafından soyut bir sistem işlevselliği üzerinde yeterince açık bir şekilde belirtilmemesi, ihtiyaç duyulan sistem ile teslim edilen arasında oluşan farkın ancak teslim tarihinde anlaşılabilmesine neden olmakta ve bunun yeniden düzenlenmesini yapmak bir hayli masraflı ve zor olabilmektedir. Bu durumun temel nedeni, eski tarz yazılım modellerinin geleneksel yöntemleri ve hiyerarşik kontrol yapısı, geliştirme aşamasında günümüz uygulama geliştirme gereksinimlerinin kolayca anlaşılıp, gerekliliklerin yerine getirilmesine izin vermemesidir. Eski yazılım sistemlerine ve gelişmelerini eski sistemlere dayandırmış kurumlardaki bu durum, onların yalnızca yazılım konusunda geri kalmasını sağlamakla kalmamakta, aynı zamanda hızlanan teknoloji çağında müşterilerine yeterli teknolojik hizmeti ortalama hızda bile verememesini meydana getirmektedir. Bunun sebep olduğu asıl sorun ise, büyük ama eski yapılı şirketlerin piyasada yeni teknolojilere sahip küçük şirketlerle bile hizmet hızı ve kalitesi anlamında rekabet edememeye başlamasıdır (Gruver ve Mauser, 2015; Hughey, 2009).

Çalışmanın önceki bölümlerinde anlatılan gerek esneklik yapısı gerek ise müşteri ile birlikte çalışmaya uygunluğu açısından Çevik model, Şelale modele göre birçok yararlı özelliklere sahiptir. Ayrıca Çevik model, katı olmayan disiplin yapısı ile birlikte, projenin yapısına bağlı olarak birtakım tasarruflar sağlayabilmektedir. Öncelikle yazılım geliştirme ve kod dağıtımı için gerekli olan detaylı araştırma ve planlama evresi bu modelde ya hiç yoktur ya da çok az bir miktarda mevcuttur. Bunun yerine bu adım daha çok geliştirme sürecinde gerçekleştirilmektedir. Yazılım ekibi ve müşteriler arasındaki kişisel iletişim, yazılımın müşteri tarafından ayrıca gözlenip test edilmesine olanak sağlamaktadır. Belge gibi işlem verimliliğini düşüren durumlar verimi düşürecek derecede yoğun değildir. Bunların yanı sıra yazılım geliştirici personel ile işletmeci, gerektiğinde farklı roller ile bir araya gelerek projenin ilerlemesine katkı sağlayabilecek esnekliktelerdir (Mircea ve ark., 2013).

Çevik model, esnek yapısı ile aynı zamanda hata oranını ve proje sonunda beklentileri karşılamama durumunu en aza indirgeyebilen bir modeldir. Önceki bölümlerde değinildiği üzere Şelale modelin sahip olduğu öngörülemez problemler riski Çevik modelde çözümsüz değildir. Bunun temel sebebi, daha önce bahsedildiği üzere müşteri geri bildirimlerinin sıklığına bağlı olarak gerçekleşen ve Şelale modele göre daha sık oluşan kod dağıtımları ve her dağıtım sonunda yapılan testlerin sıklığıdır. Aynı zamanda çevik modeli özel yapan durum, dağılan kodların henüz proje tamamlanmadan müşteri karşısına çıkmasıdır. Bu nedenle gerek kod yazılımı ve uygulama gelişimi sırasında, gerek ise müşterilerin iş kolunda var olan hızlı dönüşümlere ayak uydurması açısından projenin teslim tarihinde yaşanabilecek hata oranlarını veya memnuniyetsizlikleri en aza indirilir (Mircea ve ark., 2013).

Çevik modelin sahip olduğu en büyük artısı, günümüz sahip olduğu farklı metodolojiler ile günümüz koşullarındaki her duruma uyum sağlayabilmesidir. Çalışmanın önceki bölümlerde görüleceği üzere, her bir model, bir önceki modelde duyulan eksiklikler ile oluşturulmuştur. Ancak Çevik model, tüm bu eksiklikleri tamamlayıcı ve her bir gelişim evresini birleştirici bir manifesto ile başlamış ve sonradan ihtiyaçlara göre oluşturduğu metodolojiler ile gelişimini tamamlamıştır. Bu

neden ile diğer modellere göre daha kapsayıcı bir disipline sahiptir. Örneğin, günümüzde çoğu yazılım ihtiyacı olan kuruluşların talepleri, bir seferliğine bir teslimatı içerecek derecede basit değildir. Bu nedenle bu kuruluşların bazıları sürekli güncellemeye ihtiyaç duydukları ve sonu olmayan bir model ihtiyacını duymaktadırlar. Şelale modeli gibi eski SDLC modellerinde bunun mümkün olmadığını görürken, Çeviğin bir metodolojisinin tam olarak bu sürekli gelişim ihtiyacına karşılık verir nitelikte oluşturulmuştur. Bu nedenle sahip olduğu metodolojiler sayesinde Çevik model, günümüz uygulama geliştirme ihtiyaçlarına karşılık verme konusunda karşılaştırılamaz bir üstünlüğe sahiptir.

Sonuç olarak Çevik uygulama yönetim metodolojileri, sahip olduğu kısa zamanlı proje geliştirme döngüleri, artan dağıtım ve kontrol sıklığı ve pazara daha hızlı entegre olabilme özellikleri sayesinde eski sistemlere kıyasla bugünün ihtiyaçlarına cevap verebilir niteliktedir. Bahsedilen bu üç temel faktör, genel yazılım geliştirmelerinde ve verimliliğin artmasına neden olur. Verimliliğin yanında uygulama oluşturmada gerçekleşen bugünün müşteri talepleri, sistemin disiplininden çok esneklik özelliğine önem verir hale gelerek evrimleşmiştir. Bahsedilen bu evrimleşme, yalnızca yeni oluşturulan uygulamaları değil, aynı zamanda piyasa rekabet koşullarına ayak uydurmak isteyen büyük ve karmaşık yapıdaki sistemleri eski disiplin anlayışlarını bırakmaya zorlamıştır. Birçok büyük şirket, hali hazırda yazılım geliştirme döngülerini DevOps disiplini altında Çevik modeli ile çözüm bularak uyarlamaktadır. Bu şirketler arasında Amazon, Netflix, Hedef, Walmart, Nordstorm, Facebook, Adobe, Sony Pictures bulunmaktadır (Null, 2015).

4.2 Çevik ve Şelale kullanan sistemlerin kalitesi

Uygulamaların veya yazılım projelerinin kalitelerini ve sağlığını incelemek, aslında göreceli olarak tabir edilmesi gereken bir süreçtir. Daha önce bahsedildiği üzere her iki yazılım projesi disiplininin de belirli pozitif ve negatif yanları mevcuttur. Bu konuda değerlendirmede bulunmak için güvenilir bazı kurumların yaptığı incelemeleri takip etmek bize daha doğu bir değerlendirme sunar. Bunun için CAST araştırma laboratuvarları olarak bilinen kurumun beş yapısal kalite özelliğini temel alarak iki yılda bir gerçekleştirdiği yazılım sağlığı ve eğilimleri isimli rapor bu konuda yardımcı olacak en güzel kaynaklardan birisi niteliğindedir. Bu bahsedilen beş farklı kalite özellikleri; sağlamlık, güvenlik, performans, esneklik ve aktarım hızı olarak belirlenmiştir. 2015'de yapılan bir çalışmada Çevik ve Şelale modelleri karşılaştırmalı olarak kullanılmıştır. Uygulama kapsamında Çevik model kullanan 57 Java-EE modeli, Şelale model kullanan 60 Java-EE modeli, Çevik ve Şelale modelinin ikisini birlikte karışık halde kullanan 46 Java-EE modeli ve herhangi bir SDLC model içermeyen 21 Java-EE modeli kullanılmıştır. Belirli testler sonucunda projelerdeki hata oranlarını temel alarak oluşturulan rapora göre SDLC modelleri arasında önemli farklar gözükmektedir (Curtis ve ark., 2014).

Modeller arasındaki en büyük farklar sağlamlık ve esneklik alanlarında gözlemlenmiştir. Her iki faktörde de uygulanan HoC (Random Software Testing) testi sonucu her iki uygulamanın tek başına üstünlüğünden ziyade ikisinin karışımı olan uygulamaların üstünlüğü ortaya çıkmıştır. Aynı şekilde iki modelin karışımı, uygulanan güvenlik testlerinde de üstünlüğe sebep olmuştur. Performanslar arasında en küçük farklar performans ve aktarım kalitesi testlerinde görülmüştür. Ayrıca raporlar yalnızca Çevik veya yalnızca Şelale modelini kullanarak oluşturulmuş projelerde kod yapısında veya mimarisinde belirli bir fark bulamazken, genel anlamda en düşük kalite belirli herhangi bir yapısı olmayan sistemlerde gözlemlenmiştir (Curtis ve ark., 2014).

Bahsedilen bu modelden yola çıkarak, tek başlarına Çevik ve Şelale modelleri kendilerine göre belirli avantajlar ve dezavantajlara sahip olsalar da aslında ikisinin karışım olarak ortaya konduğu modeller, daha etkili ve az riskli bir proje disiplinini uygulama geliştirme alanında müşterilere ve yazılımcılara sunmaktadır. Ancak performans gereği iki modelin birleşiminden gelen verimlilik daha fazla olsa da, atlanılmaması gereken durum DevOps disiplinin Şelale modeline uygun olmamasıdır. Bu nedenle verimliliği bahsedilen ölçütlerden ziyade proje esneklik yapısına odaklanarak yeniden değerlendirdiğimizde Çevik model diğerlerine göre en uygun ve verimli sonucu vermektedir. Bunun yanı sıra hali hazırda Çevik modelin sahip olduğu farklı metodolojiler, eski sistemlerin bir karışımı niteliğindedir. Ayrıca Şelale modelinin günümüzde popüler uygulama geliştirme disiplinlerinden olan DevOps disiplinine uygun olmamasını anlamak için DevOps'un ne olduğunu anlamak, başlangıç olarak önemli bir gerekliliktir.

BÖLÜM 5. DEVOPS

5.1. DevOps Tanımı

Geçmişte geleneksel kod sistemleri, daha önce bahsedildiği üzere çalışmaların sabit ve katı bir disiplin içerisinde izlenmesini benimseyen bir kültüre sahipti. Ancak bu durum müşterilere sonuç olarak istediklerini hatasız olarak verilmesini sağlasa bile süreç içinde fikir değişikliği durumunu imkansızlaştıran bir yazılım metodolojini ortaya çıkarmaktaydı. Bu durum, proje esnekliğine yer verilmemesi durumunu ortaya çıkarıyordu. Bahsedilen bu sistemde her ekibin kendine ait görevleri ve kırmızı çizgileri vardı. Bu durum ekipler arası iletişimin ve iş birliğinin minimuma indirgenmesine neden olmaktaydı. Bu sorunun çözümü niteliğinde geleneksel kültürlerin aksine, geliştirme ve operasyon ekipleri aynı projenin aynı adımında, test, dağıtım ve yönetim aşamalarının hepsinin kapsandığı, SDLC'nin tamamında birlikte çalışan gruplar haline getirdi. İsminden de anlaşılacağı üzere uygulama geliştiricileri (Developers) operasyon yöneticileri (Operations) ile birleşerek projeleri farklı bakış açılarından görülebilen ihtiyaçlar çerçevesinde ilerletmeye başladırlar. Bu disiplinde eski sistemlerin kontrollerini kaybetmeden daha verimli yöntemler ile yeni sistemleri daha hızlı ve esnek bir yapıda oluşturulmasına olanak sağlanmıştır (Garinchaud, 2012).

Proje yaşam döngüleri alt başlığında yer alan proje oluşturma ve yaşam süreçlerinin belirli tanımlamalarını yapmamızın ardından, DevOps'un günümüz uygulama oluşum süreçlerindeki öneminden bahsedebiliriz. Öncelikle DevOps ve bundan önce tanımlanmış olan modeller, günümüzde birbirlerine sıkça karıştırılmaktadırlar. Çevik model ile proje takibi denildiğinde akla DevOps, DevOps denildiğinde de akla genellikle Çevik model gelebilmektedir.

Bunun temel nedenlerinden birisi Çevik modelin günümüzde yakaladığı popülerliği DevOps ile birlikte sürdürmesi iken diğer bir neden Çevik modelin hali hazırda sahip olduğu geniş tanımlardır. Yani diğer bir söylem ile iki tanımlamada aynı koşullar altında ve aynı verimliliği sağlamada kullanılan, genelde birbirlerinden pek ayrılmayan iki model olarak karşımıza gelmektedir. Ancak bu durumda aradaki fark aslında zor gözükmesine rağmen kolay ve anlaşılabilirdir (Garinchaud, 2012).

DevOps bir projenin oluşumunda izlenen disiplin anlayışı iken, Çevik, o projenin nasıl ilerlemesi gerektiğini belirten bir modeldir. Diğer bir anlatım ile DevOps projenin gelişiminde kimlerin yer alacağını anlatan bir metodoloji iken, Çevik, projenin hangi adımlar izlenerek geliştirileceğini gösteren bir haritadır. İki tanım arasında bahsi geçen karmaşanın temel nedeni, Çevik modelin manifestosunda yer alan geliştirici ve operasyon birimlerinin birlikteliği adına sıkça yapılan vurgudur. Bu nedenle bu çalışmada bu ayrımı doğru yapabilmek adına proje yaşam döngülerini oluşturan modeller, DevOps anlatımının öncesinde tanımlanmıştır.

Literatürde Şelale sistemlerinde olduğu gibi, DevOps sistemlerinde de birden fazla tanımlamalarla karşılaşılmaktadır. Bunlardan en yaygın ve kullanımda olanı DevOps'u tasarım ve geliştirme sürecinden üretim desteğine kadar tüm sistem döngüsü boyunca işletme ve geliştirme mühendislerinin ortak çalışması olarak tanımlanmaktadır. Genel tanımlamaların ardından aklımıza gelebilecek ilk soru, neden DevOps gibi bir sisteme ihtiyaç duyulduğudur. Uygulamaların gelişme aşmalarında sorumluluğu yalnızca bilgi teknolojilerinde çalışan birimlere vermek ve kontrolü yalnızca yöneticilerin devralması yerine, DevOps anlayışı DevOps'u oluşturan Geliştirme ve Operasyon olarak tasvir ettiğimiz bu iki grubun yazılım geliştirme süreçlerinde birbirlerine bağlılığını benimser. Eski sistemlere göre daha sık yapılan kontroller ve iki takım arasındaki görüş alışverişleri projelerin daha sağlıklı ve verimli bir sonuca ulaşmasını sağlamaktadır. Bu disiplin ile daha önce belirtilen, eski sistemlerde sıkça rastlanan proje sonu hataları gelişim sürecinde fak edilerek en aza indirgenmektedir. Tanımlanan bu verimlilik artışından ziyade diğer bir önemli etmen, günümüz iş sistemlerinde ihtiyaç duyulan konulara artık eski

uygulama geliştirme disiplinlerinin cevap vermemeleridir. Günümüzde gerçekleşen yazılım anlamındaki iş kollarına bakıldığında, artık uygulamaların yıldan yıla gelişimleri değil, her gün gerçekleşmesi gereken gelişimleri ön plana çıkartılmıştır. Bu her iş kolunda böyle olmasa bile, bu tür uygulamalara sahip şirketlerin sayısı gün geçtikçe artmaktadır. Bankacılık ve e-ticaret sektörleri, bu tür uygulamaların en çok rastlandığı yerler olarak örnek gösterilebilir (Baker, 2015).

Bunun yanında DevOps daha kısa üretim veya geliştirme döngülerinde daha yüksek kalite koduna ulaşmaya çalışan kuruluşların karşılaştıkları zaman kısıtlamaları ve proje tamamlama baskılarını ortadan kaldırmayı hedefleyerek tamamen kalite yönetimine ve proje gelişiminin işlevselliğine odaklanır. Proje, en başta tanımlandığı şekilde yazılmaktan ziyade, zamanla oluşan tanımlar çerçevesinde bir gelişim döngüsü ile ilerler. Sonuç olarak kod tamamen test edildiğinde ve farklı adımlarda belirtildiği gibi tasarlandığı ve çalıştığı sürece hayata geçirilir ve proje müşteriye teslim edilir (Baker, 2015).

Günümüzde DevOps proje disiplinine en uygun olarak görülen ve yaygın olarak tercih edilen SDLC modeli Çevik modeldir. Çevik model disiplininin tam zıttı olarak adlandırılan diğer bir yaygın model ise daha önce tabir edildiği üzere Şelale modelidir. Bu nedenle çalışmanın giriş kısmında da belirtildiği üzere DevOps disiplinine geçişlerde bizim yeni olarak adlandırdığımız model Çevik, eski olarak adlandırdığımız model ise Şelaledir. DevOps geçişlerinde yaşanan zorluklar ve yapılan karşılaştırmalar bu yaygın kullanım alışkanlığı temel alınarak yapılmıştır.

5.2. DevOps Ortamına Dönüştürme

Ürün kalitesini iyileştirme, çağın gerektirdiği esneklik yapısına sahip olma ve yenilenmeyi hızlandırma konularındaki başarıları sayesinde DevOps metodolojisi bugün popülerliğini arttırarak piyasada bilinirliğini arttırmaya devam etmektedir. Daha önce değinilen birtakım avantajlarından ötürü kurumsal firmalar artık DevOps yazılım modellerine dönüşme kararları almaya başlamışlardır. Özellikle zamanın daha hızlı geçtiği ve gelişmelerin daha kısa zaman aralıklarında gerçekleştiği günümüzde DevOps yaklaşımı projelerin başarısızlık oranlarını düşürmede de aktif rol oynamaktadır (Bentley ve ark., 2015; Bradley, 2015).

SDLC'nin Çevik olarak adlandırılan modeli, DevOps geçişlerinde ve disiplininde kullanılan en yaygın modeldir. Sıralı bir tasarım ve proje süreci yerine Çevik modeli, birden fazla proje yaşam döngüsü ile artan ve süreç içerisinde kontrol ve müdahaleye elverişli bir sistem izler. Doğrusal olan Şelale modelinin aksine Çevik, planlama, ihtiyaç analizi, tasarım, kodlama, birim testi ve kabul testi gibi çeşitli alanlarda aynı anda çalışabilen çapraz birimleri içerir. Çevik modelde çapraz süreçler ile ilerleme sağlanırken aynı zamanda daha kısa süreli görüşmeler ile de müşterilerin talepleri kısa zaman birimlerinde incelenir. Bu sonuçta da projenin işlevselliği kendini korumaya ve güncel tutmaya devam eder. Bu sistemler bütününün amacı kodun süreçten ziyade örgütsel iş birliğine odaklanan küçük modüllerde hızlı bir şekilde üretilmesidir. Çevik model takım çalışmasına ve çapraz eğitime olanak sağlamak ile birlikte, sistemin işlevselliğini geliştirmede ayrıca katkı sağlar. İşlevsellik konusunda proje odaklı hızlı geri bildirim vardır. Kod sorunları çok çabuk bulunur ve süreç içerisinde çok çabuk düzeltilebilir. Daha az dokümantasyon söz konusu olmak ile birlikte sistem adına yararlı olduğu inanılan şekilde beklenmedik geliştirmeler, modelin esnekliği sayesinde sisteme dahil olabilir. Sahip olduğu tüm bu özellikler ile Çevik modeli bir nevi Şelale modelinde yaşanan sorunların bir çözümü olarak DevOps disiplininde karşımıza çıkmaktadır.

Bahsedilen bu avantajlarına rağmen ne yazık ki DevOps dönüşümleri için ortaya atılmış kesin bir rehber bulunmamaktadır. Aksine DevOps dönüşümleri özellikle eski yapılı sistemler için büyük zorluklar teşkil etmektedirler. Daha önce geçiş yapmak ve tüm ekibi farklı olan bir metodolojiye taşımak isteyen kuruluşlar bu konuda başarısızlık yaşamışlardır. ING bankasının bu konuda 2010'da başlattığı dönüşümden 2014'te başarısızlık ile çıkması, bu durumun güzel örneklerinden birisidir. Bir gecede bütün sistemi DevOps disiplinine geçirmek yerine yapılması gereken durum küçük ölçekli projeler ile geçişin sağlanmasıdır. Bu şekilde hem ekipler disipline alışmakta zorluk çekmez, hem de hali hazırda yapılan işlerde yaşanan aksaklıklar en aza indirgenmiş olur (Ambler, 2003; Bentley, 2015).

DevOps disiplinindeki uygulamalar eklenebilen veya çıkarılabilen güçlü bir kod tabanı oluşturmaya dayandığından, DevOps için uygulamaları seçerken eski bir uygulamanın yeniden yapılandırılabilir olup olmadığı dikkate alınmalıdır. Yeniden düzenleme ile burada tanımlanan durum, kodun işlevini ve servisini değiştirmeyecek şekilde yazılım yapısını değiştirmektir. Bu süreç, kaynak kodun revizyonu ile meydana gelmektedir (Fowler, 2018).

Bahsedilen eski sistem yapılarına sahipken başarılı geçişler yapmış bir şirket olan IBM, kuruluşların DevOps disiplinini benimsemesi ve bu geçişi gerçekleştirmesi için altı farklı geçiş yolu önermektedir;

1. Sürekli iş planlaması yapan ve organizasyon yapısından emin olan bir BT takımının değişen ihtiyaçlar çerçevesinde geçiş yapması
2. DevOps takımları ve Çevik metotlarını kullanarak geçişleri sağlama
3. Otomatikleştirilmiş kod testleri ile sistemi sürekli test etmek
4. Kodun sürekli bir biçimde dağıtımını sağlamak
5. Sürekli kontrol
6. Sürekli optimizasyon (Edler ve ark., 2014).

Önerilen yollardan ilki organizasyonların kendi yapılarından emin olarak yapması gereken sürekli iş planlamasıdır. Bu iş planlamaları kurum içindeki verimliliği arttırmanın yollarını arayan yöneticilerin yaptığı baskılardan gelmektedir. Bahsedilen yönetim baskılarının BT ekipleri üzerinde olan kısımları genellikle sistem masraflarının azaltılması, yeni teknolojilerin yardımı ile karlılığın veya farklı iş kollarının oluşturulması, iş süreçlerinin daha teknoloji odaklı entegrasyonunun sağlanması, yaşlanan ekip üyeleri sorunu ve şirketin sahip olduğu özel yapıların kopyalanmasının önüne geçilmesi gibidir.

Eski yapılı bir sistemi esnek yapılı bir Çevik sistemine dönüştürmek için dönüşümü düşünülen yapının özelliklerini iyi analiz etmek gerekir. Her kurumun ve kurumun sahip olduğu sistemlerin kendilerine özel durumları vardır ve geçişin sağlanması bu durumların analiz edilip bunlara göre oluşturulacak stratejiler ile gerçekleşebilir. Herhangi bir geçiş yapmadan önce bu planlamaların gerçek anlamda yeterli ve verimli olması, yapılacak geçiş süreçlerinde farklı bir engel ile karşılaşma sorununun önüne geçebilmektedir. Bu konuda projeler tamamlamış ve alanda tecrübeye sahip bir kurum olan Rex Laster, Çevik uygulama planına geçecek her kurumun aşağıda belirtilen maddeleri mutlaka gözden geçirmesi gerektiğini belirtmektedir.

1.Organizasyon yapısı neye benzemektedir. Büyük bir organizasyon mudur yoksa küçük çaplı bir organizasyondan mı bahsedilmektedir? Dinamik yapıda mıdır yoksa esnek olmayan bir yapıya mı sahiptir? Odaklandığı durum maliyet midir yoksa projenin katacağı değer midir?

2.Organizasyon tam olarak nerede Çevik modele geçmek istemektedir veya bu ihtiyacı duymaktadır. Organizasyon herhangi bir Şelale projesi yürütüyor mu? Daha önce Çevik ile başlanmış ancak başarısız olunmuş durumlar mevcut mudur?

3. kuruluşun verimli bir değişim hareketini yönetme kapasitesi nedir? Denenmiş olan bir geçiş yöntemi varsa insanlar bu geçişi benimsemiş midir? Benimsemediyse geçiş projesinin hangi adımında direnç ile karşılaşılmıştır?

4. İş yapısı ve organizasyonun iş yapısı ile ilişkisi nedir? Müşteri yapısı nasıldır? Projenin yapısı veya yeni sistemlerin yapısı müşteri yapılarını desteklemekte midir?

5. Çevik değişimi kuruluşun herhangi bir alt projesinde başlatıldı mı? Başlatılan geçişin kurum kapsamında büyüklüğü nedir (Lester, 2013)?

Daha önce bahsedildiği üzere, bu geçişin zorluğundan dolayı organizasyon yapısına ve geçişin sağlıklı olmasına yardımcı olmak adına bahsedilen sorular erkenden cevaplanarak geçişe hazırlanan rapor ile devam edilmelidir.

Geçişteki kritik noktalardan birisi de DevOps'un sürekli devam eden test süreçlerinden meydana gelmesidir. Testler projelerin bölümlerinin bitimleri ile başlamak yerine henüz gelişme aşamasında başlarlar ve sürekli olarak devam ederler. Bahsedilen bu otomasyon satıcı tarafından oluşturulmuş ticari araçlardan gelebilir, açık kaynaklı kodlardan gelebilir veya kurum içi özel olarak hazırlanmış araçlar tarafından oluşturulabilir (Novak, 2014).

IBM'in geçiş konusunda hazırladığı maddelerden bir diğeri de sürekli olan dağıtım süreci ile ilgilidir. Şelale modelde olanın aksine Çevik model, kodun her aşamada ve her test süreci sonunda dağıtımını öngörür ve buradan gelecek geri bildirimler veya hata raporuna göre, projeyi kusursuz hale getirebilmek adına yeniden yapılandırarak işlemlerine devam eder. Bu durum aynı zamanda eklenmesi gerektiği tüketici yorumları tarafından belirlenen kısımların yeniden yazılarak uygulamaya eklenmesi ve bunun yeniden dağıtılması arasındaki süreyi yüksek oranda ve verimli bir şekilde

düşürür. Bu durum Çevik sistemlerine sahip olduğu en önemli özellik olan esnekliği kazandıran durumdur. Ancak buna zıt olarak bazı kuruluşlar bu sürekli dağıtım durumunun yeterli testler sağlanmaması durumunda operasyona bağlı riskleri beraberinde getirdiğini vurgulamaktadır. Bunun önüne geçilmesi adına yapılan testlerin disiplini ve işlevselliği özel olarak incelenerek bu sistemin uygulanmasına geçilmelidir (Novak, 2014).

Son olarak IBM'in belirttiği DevOps geçiş stratejilerinde yer alan önemli maddelerden birisi de sürekli gözlem ve kontrol yapılmasıdır. Daha önce belirtildiği üzere Çevik sistemlerinin popülerliğinde büyük rol oynayan esneklik özelliği genel olarak sürekli dağıtım ve sürekli geri bildirim almaktan ve geri bildirimlere göre projeleri yeniden dizayn ederek yeni dağıtım ile eksiğin veya hatanın giderilmesinin hızlı bir şekilde yapılmasından geçmektedir. Ancak yapılan bu yöntem belirli riskler içerebilmektedirler. Testlerin sürekli olması aslında bir nevi bu risklerin düşmesini sağlarken testlerin otomatikleşmesi, beklenen yerlerdeki belirli hataları ortadan kaldırmaya yaramaktadır. Bunun dışında beklenmeyen bir yerden kaynaklanabilecek herhangi bir hata durumunda çözümü hızlı bir şekilde oluşturmak ve müşteriye sunmak, hali hazırda işleyen bir iş kolu için çok büyük önem arz etmektedir. Bu nedenledir ki bu süreçte gözlem, riski neredeyse sıfıra indiren son güvenlik adımını uygulamaktadır (Cois, 2014).

5.2.1. Yazılım Mimarisinin Yeniden Yapılandırılması

Daha önce bahsedildiği üzere yazılım mimarisi veya proje oluşturma disiplininden kaynaklanan hatalar veya eski çözüm yollarının etkisini kaybederek günümüzde yerlerini yeni uygulamalara bırakması beraberinde bir değişimin gerekliliğini getirmektedir. Bu konuda çeşitli çalışmalar ve bu gerekliliğe sahip şirketlere danışmanlık veren çeşitli yazılım mühendisleri mevcuttur. Bunlardan en bilinen analizci ve Çevik model uzmanı Martin Fowler, yeniden düzenleme olarak tanımlanan bu uygulamayı mevcut sistemlerin dış yapısını değiştirmeden iç yapısını

değiştirmek olarak tanımlamaktadır. Yeniden düzenleme olarak adlandırılan bu iyileştirme sürecinin bilinen faydaları; uygulama kodunun uzunluğunun azaltılması, yedeklemelerin azaltılması, karmaşıklığın azaltılması, iyileştirilmiş kod anlayışı, iyileştirilmiş tasarım ve daha sonra yeniden kullanılabilecek, değişikliğe açık kodların üretilmesidir (Fowler, 2018).

Tanımlanan bu yeniden yapılandırmanın ikinci bir yolu ise uygulamanın işlevini değiştirmeden yazılımın temel yerlerinde değişikliğe gidilerek gerçekleştirilen değişimdir. Bu yol yazılım dünyasında soyutlama olarak adlandırılır. Bu yolun temel amacı, işlevselliği koruyarak daha az karmaşık ve verimli bir kod mimarisine ulaşmaktır. İsim olarak soyutlama adını almasının temel nedeni, bu çözümde kodun kendine ait olan karakteristik özelliklerinin değiştirilerek veya kaldırılarak kodu belirli bir karaktere ait olmayan esnek yapıya sokmaktan gelmektedir. Bu da daha önce bahsedilen esnekliği önleyici ve sonradan kullanımı imkansızlaştıran durumları ortadan kaldırarak yazılımcıların bir sonraki değişiklikleri kolayca uygulaması anlamına gelmektedir (Gruver ve Mouser, 2015).

5.2.2 Otomasyonun uygulanması

Eski yapılarda kullanılan Şelale modeli, sistemli uygulamalarda ve geliştirme projelerinde kullanılan ve tamamlanması uzun süren test sistemleri yerine Çevik model sistemleri otomatikleştirilmiş ve sürekli hale getirilmiş test sistemlerini kullanmıştır. DevOps test otomasyonlarındaki temel amaç, geliştirmeden son ürüne kadar olan süreçteki farklı bölümlerde yer alan kaynak kod aracılığı ile bir takım ortak senaryodan veya durumlardan yararlanmaktır. Bu ortak durumların kullanımının mümkün olmadığı yerlerde ise değişkenlerin değiştirilebileceği şekilde tasarım yapmak gerekmektedir. Bu ana amaca uygun kalınmadığı takdirde test komut dosyaları, başarılı bir test çıkarmak ve sürdürmeyi imkansızlaştıracak kadar büyük hale gelir. Bu nedenle test otomasyonları DevOps için çok büyük öneme sahiptir. Otomatik test sistemlerine geçişte başlangıç olarak izlenmesi gereken yol yönetim ve

teknik liderlerin eski sistemlerinde otomatikleştirmenin gerekli olduğu yerleri incelemektir. Bu aynı zamanda otomatikleştirmenin nasıl gerçekleşeceğini tanımlamak ile de devam etmelidir. Otomatikleştirmenin yapılabileceği birkaç önemli yer mevcuttur. Bu şekilde projelerin hata paylarının düşmesinin yanı sıra, kod teslim kalitesinin de artışı sağlanır (Gruver ve ark., 2012).

Bahsedilen testlerin uygulaması için en iyi üç adet uygulama vardır. Bunlar;

1-) Emek yoğun ve tekrarlanan testlerin otomatikleştirilmesi

2-) Hataların sıklıkla bulunduğu testlerin otomatikleştirilmesi

3-) Dağıtımın otomatikleştirilmesi.

Bahsedilen bu noktalar, DevOps kullanımında otomatikleştirmenin önemli olduğu yerler olsa da eğer kuruluşun eski mimarisi birbirine bağlı esnek olmayan bir yapıyı içermekteyse, otomatikleştirmenin yoğun olarak yapıldığı bahsedilen yerlerde otomatikleşmeyi yapmak zorlu bir süreç haline dönüşebilir. Bu gibi kodlama ve dağıtımın doğrudan ilgili olduğu kısımların yanında kuruluşlar kodlama ve dağıtımın direkt olmadığı zamanlama, güvenlik formları ve uygulama profili formları gibi yerlere de testler için göz atmaları gerekir. Bu alanlar, bahsedilen karmaşık mimariye sahip kuruluşlar otomatikleştirmenin başlangıç yerleri olabilirler. Tommy Mouser ve Gray Gruver, bu konuda özellikle eski yapılı sistemlerin otomasyonlarında testlerin önemini vurgular ve zorluğunu kabul ederler. Bunun yanında DevOps'da gereken Sürekli dağıtımın iyi çalışması için eski ve karmaşık yapıdaki sistemlerin problemleri hızlı yakalayabilen otomatik bir yapıya sahip olmaları gerektiğini belirtirler. Bunun aslında çoğu kurum için birden gerçekleştirmenin neredeyse imkânsız olduğunu belirtmekle birlikte, önceliğin ulaşılabilir hedeflerde kalınmasını ve daha sonrasında çalışma ve geçiş istikrarının arttırılmasını tavsiye etmektedirler (Gruver ve Mouser, 2015).

Bunların yanı sıra DevOps test süreçlerindeki otomasyon fikri aslında bazı yazılımcılar tarafından eski sistemin yerini tutmayacağı endişesini taşımaktadır. Testlerin tüm hataları görmekte zorluk çekebileceği veya yeni oluşabilecek hataları kaçırabilecekleri gibi iş disiplini ve sorumluluklarına zarar verebilecek endişeler konu hakkında ileri sürülmüştür. Ancak burada anlaşılması gereken durum otomasyon süreçlerinin testleri düzeltmek yerine yalnızca belirli kısımları otomatik olarak çalıştırarak hataları göstermesi, DevOps otomasyonunun gerçekleştirdiği asıl eylemdir. Ancak test otomasyonu hakkında bahsedilmesi gereken önemli bir sorun, DevOps için hazırlanmış otomasyon araçlarının eski sistemlere bazen uymuyor olmasıdır. Özellikle DevOps'un geçiş aşamasında olan kuruluşlarda bu durum gözlemlenmektedir. Bu nedenle iki farklı proje veya kurum türüne uygun test araçlarının seçilmesi bu problemin azalmasını sağlayacaktır (Riley, 2014*;* Liu ve ark., 2014).

5.3. DevOps'da Karşılaşılan Güvenlik Endişeleri

DevOps günümüz için uygulanması uygun bir sistem olarak gözükse de birtakım sorunları içinde barındırmaktadır. DevOps metodolojileri daha önce bahsedildiği üzere geleneksel ve eski BT sistemlerini bozar ve organizasyonların eskisinden daha fazla güvendiği otomasyona dayalı yeni sistemler getirir. Ancak atlanılmaması gereken önemli hususlardan birisi bu otomatikleştirilmiş araçların oluşturabileceği güvenlik araçlarıdır. Otomatikleştirilmiş test araçlarına duyulan aşırı güven, bu araçların yazılım korsanları tarafından sisteme girmedeki bir kapısı haline gelebilmektedir. Bunun temel nedenlerinde birisi, araçlar hakkında yeterli uzmanlığa sahip olmayan bilişim çalışanlarının otomatikleştirilmiş test araçlarını uygulamanın gerek olmayan yerlerinde dahi kullanmasıdır. Özellikle büyük ölçekli teknoloji kuruluşları açısından bu durum büyük önem teşkil etmektedir. Güvenlik kaygısındaki diğer bir unsur ise otomatikleştirilmiş test araçlarının açık kaynakları kullanmasıdır. Günümüzde kullanıma açık birçok güvenilir kaynak olduğu bilinse bile yapılan araştırmalara göre bu bilinen kaynaklarından 16'sından birisinde güvenlik açıkları tespit edilmektedir. Diğer bir söylem ile DevOps metodolojisine ait olsun veya

olmasın, açık kaynakların bir kısmında güvenlik açığı bulmak imkânsız değildir. Bu nedenle bilişim ekibinin bu konuda bilgilendirilmesinin yanı sıra gerekli eğitimin verilerek konu hakkındaki hassasiyetin giderilmesi gerekmektedir. Bu durum aslında yapı içerisinde takımlardan çok sisteme dayalı güvenin artmasını ve bu artışın aslında sistem riskini beraberinde getirmesini sağlamaktadır. Sonuç olarak kurumlar, bilgisayar korsanları adına daha erişilebilir bir hedef haline gelebilmektedir. Herhangi bir şekilde bilgisayar korsanları sisteme eriştikten sonra güvenlik duvarlarını değiştirmek, hesap eklemek, üretim sistemlerine erişimi izin vermek, veri çıkarmak, fiyatları değiştirmek ve hassas güvenlik açığı bulunan yazılımları yüklemek gibi istedikleri herhangi bir yapılandırmayı değiştirebilirler. Bunları belirten Storms, asıl sorunun sistem açığı değil bu açığın boyutunun tam olarak belirlenememesi ve bu konuda yeterli adımların veya planlamaların yapılmamış olması olarak tanımlamaktadır. Ancak güvenlik açıklarının nedenleri, bize Çevik modelde bu konuyu nasıl çözebileceğimizi göstermektedir. Daha önce belirtildiği gibi personelin eğitimi, açık kaynaklı kodlara önlem ile yaklaşılması ve başarılı bir güvenlik planlaması, bahsedilen ve savunulan bu güvenlik sorununu ortadan kaldıracaktır (Rubio, 2015; Mohamed, 2016).

Güvenlik konusunda DevOps'un getirdiği endişelerin yanında bir diğer problem denetim mekanizmasıdır. Eski sistemlerde olan bilişim denetim ekipleri, DevOps disiplininde denetimi yeterince sağlayamadıklarını düşünmektedirler. Bunun temel sebeplerinden birisi yazılım geliştirici ekibin ürettiği kodları henüz proje tamamlanmadan ve tam olarak hazır olmadığı halde dağıtmasıdır. Ancak bu endişe, önceki güvenlik sistemlerine göre daha zayıf argümanlara sahiptir. DevOps disiplininin doğru bir şekilde uygulandığı durumlarda hali hazırda her tamamlanan kısımda uygulamanın dağıtımından önce otomatikleştirilmiş bir test süreci mevcuttur. Belki bu durum eskisi kadar zaman harcamamak adına geniş denetimlere olanak sağlamamaktadır ancak yine de denetimi imkansızlaştıran bir durumun yüksek derecede riskleri getirmesi söz konusu değildir. Bunun yanında birçok kuruluş, açık kaynak kullanımı yapan ve bu sebeple güvenlik risklerin arttıran DevOps araçlarına güvenmektedir. Bu güvenlik zafiyeti genel olarak açık kaynak kullanımındaki artış ile sebeplendirilmektedir.

Normalde bu açıkları ve hataları kontrol etmek adına Bilişim teknoloji denetçileri sistem görev alırlar. Genel olarak bu alandaki denetçilerin kontrolleri, sistem girdileri ve çıktıları, işlem kontrolleri, yedekleme ve kurtarma planları, sistem güvenliği ve bilgisayar donanımının kontrollerini kapsar. Bu bağlamda iç kontrolörlerin en yaygın denetim kaygısı, geliştiricilerin herhangi bir denetime tabi olmadan kendi kodlarını dağıtabilmeleri ve bu nedenle oluşan denetimsiz yazılımların üretime geçme riskidir. Bu konu ele alındığında eski sistemlerde işlenen yapısı ile BT denetimi, DevOps kültürüne göre de uyarlanıp yeniden şekillendirilmelidir. Bu bağlamda sonuç olarak DevOps'un benimsenmesi, bahsedilen onca faydasının yanı sıra kontrol ekiplerinin riskleri anlamada zorluk yaşamasından ötürü önemli derecede güvenlik riskleri oluşturabilir (Kerravala, 2015; Ravichandra, 2015).

Bahsedilen bu kaygılar Çevik model ve DevOps disiplininde güvenliğin otomatikleştirilmesi adına çok acil bir ihtiyaç duyulduğu sonucuna ulaşmaktadır. Ancak bu durum işin içinden çıkılmaz bir hal alabilecek nitelikte değildir. Zayıf olan kısımlarda ayrıca geliştirilecek bir otomatikleşme süreci ile projelerin güvenlik zafiyeti ortadan kaldırılacak, güvenlik ve denetim ekipleri açıklar hakkında daha ayrıntılı bilgi sahibi olacak, güvenlik, projenin hedefleri doğrultusunda dengelenmiş olacak ve kod dağıtım sırasında güvenliği sağlamak adına herhangi bir yavaşlama meydana gelmeyecektir (Kerravala, 2015).

5.4. Eski Yapılı Sistemlerden DevOps'a Geçiş

5.4.1. Sistemlerin eskileşmesi

Bilgisayar sistemleri ve teknoloji uygulamaları işlerin örgütsel ihtiyaçlarına karşılık vermek ve onları bulunduklarından daha verimli hale getirmek adına desteklemek niteliğinde çalışırlar. Ancak var olan zaman kavramı ve bu zaman kavramının içinde oluşan gelişim evreleri bu durumu sabit sistemlerin kullanımındaki verim düşüklüğü olarak karşımıza çıkarır. Yapılandırılmış birçok uygulamanın ve proje geliştirme

metodolojilerinin eskime özelliği bahsedilen bu zaman kavramına bağlı olarak değişmektedir ve özellikle bulunduğumuz çağda bahsedilen bu eskime ve verimini yitirme durumu zaman kavramının ve buna bağlı olarak gelişim döngüsünün var olan yeni teknolojiler ile değişmesi üzerine daha da hızlanmış durumdadır. Bu sebepledir ki belli bir zaman süreci sonunda kurulmuş sistemler kendi etkilerini ve verimliliklerini yitirerek yenilenmeye ihtiyaç duymaktadırlar. Ayrıca günümüzde bu ihtiyacın sıklığı eskimeden ziyade zamanda oluşan evrimler ile daha fazla artmıştır. Bu da kısaca sistemlerin eskileşmesi ve evrimleşmesi anlamına gelmektedir. Örneğin, bundan 15 yıl önceki toplumun alışveriş alışkanlıkları ile bugünün toplumundaki alışveriş anlayışında bu evrimi görebiliriz. 15 yıl önce alınması planlanan bir elbise için harcanan zaman günün yarısı iken bugün bu süre teknolojinin çeşitli katkıları sayesinde yer değiştirmeden harcanan 1 saate tekabül etmektedir (Lauder ve Lind, 2006).

Eski sistemlerin betimlemesi kolay bir şekilde yapılmasına rağmen, tanımı bu kadar kolay olmamak ile birlikte literatürde farklılıklar gösterebilmektedir. Kimi kaynaklarda bu tanım kısaca hala kullanılmakta olan, değişim ve yenilenmesi mümkün olmayan ve yerine yeni versiyonlarının geçtiği eski bir bilgisayar sistemi olarak tanımlanırken, konu hakkında daha detaylı tanımlamalar başka bir uzman tarafından belirli kriterlere dayandırılarak yapılmıştır (Murphy, 2009).

- Geçmişte geliştirilmiş ancak daha sonra farklı uygulamalara geçilmesi ile unutulmuş.
- Yetersiz bilgi ve hata yoğunluğu dolayısıyla geliştiriciler tarafından değiştirilmek zorunda kalınmış,
- Hassas mimarisi sebebi ile yazılımcıların bozma korkusundan dolayı geliştirmeye çekinmesi.

Eski sistemlerin anlaşılmasında yararlanılabileceğimiz en güzel örnek, banka sistemleridir. Bundan 25 yıl öncesinde genel anlamda bankalar yalnızca kredi veren,

mevduat faizi işleten, para saklama ve transferleri hizmetlerini gerçekleştiren hizmetleri sunan kuruluşlar olarak varlıklarını sürdürmekteydi. Ancak 25 yılda gelişme gösteren gerek tüketim gerek ise yatırım anlayışları, bankaları ayda bir uğranan bir yerlerden her gün ve her anımızın vazgeçilmez bir parçası haline getirmiştir. Tüketicinin hayatında gerçekleşen bu fonksiyonel değişim, bankaların gerek hizmet yapısında gerek ise bu hizmetin taşınmasında bir nevi değişikliklere gitmesini mecburi kılmıştır. Bu nedenle 25 yıl önce bir bankanın para transferi ofisinde çalışan kişilerin yaptıkları işlemler ve bu işlemler için kullandığı yöntemler ile bugünün bankacılarında belirgin değişiklikler gözlemlenebilmektedir. Bunun en güzel örneklerinden birisini geçmişten bugüne İş Bankası'nın gerçekleştirmiş olduğu dönüşümlerden görmek mümkündür (Şuman, 2018).

Ayrıca işlevsel değişimlerin yanı sıra hizmetlerin verildiği zaman limitlerindeki değişimlerde sistemleri sürekli bir gelişim döngüsüne girmeye zorlamıştır. Geçtiğimiz günlerde Borsa İstanbul tarafından finans kuruluşları için kaldırılan öğlen seans araları, günün 1 saatinin bile bu anlamda ne kadar değerli olduğunu göstermektedir. Bu ve bu gibi örneklerden anlaşılacağı gibi günümüzde sistemler bitmez bir eskime anlayışı içine girmişlerdir ve bu nedenle eski disiplinleri uygulamaları imkânsız hale gelmiştir.

5.4.2. Neden eski sistemleri değiştirmeliyiz

Birçok problemle yüzleştiğimiz günlük hayatımızda genel olarak bu problemleri aşmak adına iyi veya kötü bir takım çözüm önerileri geliştiririz. Aslında karar aşamasında genel olarak bu çözüm önerileri güzel gözükse de sonuç aşamasında fark etmeden yaptığımız bir yanlışı çıktı olarak görmek, fark edilmeyen yanlışların daha kesinleştirici bir etkisi olduğunu karşımıza çıkarır. Bununla birlikte yazılım dünyasında da bu gibi, sonrasında geri almanın zor veya maliyetli olduğu bazı adımlar mevcuttur. Bunlar genel olarak literatürde anti-pattern olarak adlandırılmaktadırlar. Bu tarz geri dönülmesi zor veya imkânsız olan hataları şuan

için ancak geçmişte verdiğimiz kararlar yardımı ile görmemiz mümkündür. Bu açıdan bakıldığında eski yapıdaki sistem ve uygulama veya proje disiplinlerine bakıldığında karşımıza altı adet genel anlamda karşılaşılmış kötü kararlar çıkmaktadır.

1. Geçmişte yazılmış birçok uygulama genel olarak belirli sistemlere bağlı araçlar ile oluşturulmuştur. Bu da bu tür sistemlerin diğer sistemler ile iletişimini engellemektedir.
2. Birçok uygulama kendisinden farklı bir sistem ile iletişime geçemeyeceği bir dille yazılmıştır. Bu da sistemler arası iletişimi zayıf hale getirmiştir.
3. Birçok yazılım uygulamaları "monolithic" olarak kurgulanmıştır. Bu da bileşenlere ayrılıp yeniden yapılandırılmayı neredeyse imkânsız hale getirmiştir.
4. Sistemde olduğu gibi kodlar da farklı dillerde yazıldığı için aralarında iletişimi mümkün olmayabilmekte veya bu iletişim ve birleşimin oluşumu zaman alabilmektedir.
5. Eski sistemlerin bölümleri birbirleri ile sıkı sıkıya bağlı bir şekilde oluşturulmuştur. Bu nedenle herhangi bir şekilde birbirlerinden ayrılmaları veya herhangi bir bölüm üzerinde ayrık bir çalışma yapılması mümkün değildir.
6. Ana yönlendirici görevi gören "if" komutunun (guard-code) aynı bölümdeki işlemlerin farklı başlıklarında her seferinde yeniden oluşturulması, yazılım ilgili bölümdeki bağımlılıklarını arttırarak gerek değişim esnekliğini gerek yönetimini, gerek anlaşılabilir oluşunu, gerek ise ilerideki değişikliklerde yeniden kullanımını zorlaştırır. Buna kısaca kod çöplüğü veya fazlalığı denebilir (Lauder ve Kent, 2000).

Sıralanmış bu sorunlar yalnızca önümüze belirleyemediğimiz ve anti-pattern olarak adlandırdığımız sorunları getirmemekte, aynı zamanda sistemlerin değişim ve gelişim kabiliyetlerini etkilemektedir. Daha önce SDLC modellerinde değinildiği üzere, bir uygulamayı geliştirme veya değiştirme aşamasında iki seçeneğimiz

mevcuttur. Uygulamayı gerekli bakımları yaptıktan sonra güncellemek ve yeniden dağıtmak veya uygulamadan tamamen vazgeçerek farklı maliyetler üstlenerek yeni bir uygulama projesine gitmek. İlk durum, piyasada kurumların kullandığı ve tercih ettiği en kolay ve maliyetsiz yöntemdir. Ancak ikinci durum biraz daha karmaşık, maliyetli ve zor olduğundan dolayı çok fazla tercih edilmeyen bir yöntemdir. Bu nedenle ikinci durumu kabul etme mecburiyetinde kalmamak adına sistemlerimizin esneklik yapılarındaki durumu kontrol etmeli ve gelecekte yapılması gereken bir yeniliğin sistem tarafından ne kadar kabul edilip edilmeyeceğini öngörebilmeliyiz. Yukarıda belirtilen maddeler, genel olarak kurumların karşılaştığı bu sorunu ortaya koyarken, aynı zamanda bu farkındalığı sağlamak adına izlenmesi gereken yolları da kısaca bize aktarmaktadır (Lauder ve Kent, 2000).

5.4.3. Eski sistemlerin yazılım mimarisi kategorileri

Eski sistemler için genel olarak tekil bir disiplin anlayışı söz konusudur. Değişiklik göstermeyen bu yazılım stratejileri aslında projelerin karakteristik özelliklerine bağlı olarak değişiklik göstermesi gerekse de birbirlerine benzer halde planlarla tamamlanan iş süreçleri nedeni ile bu durum mümkün olmayan bir hale gelmiştir. Bahsedilen bu eski sistemlerin kullandığı uygulamalar, kullanım özellikleri ve önceliklerine göre üç farklı yapıya ayrılabilir;

Kayıt sistemleri (System of Record) – Kurumun çekirdek yapıdaki işlemlerinin işlenmesini sağlayan ve önemli verilerinin barındırıldığı alanlardır. İyi ve dikkatli düzenlenmiş süreçlerden dolayı değişim ihtiyacı düşüktür ve bu nedenle çoğu kurum için ortaktır. Genellikle yasal gereksinimlere tabidir.

Farklılaşma sistemleri (system of differentiation) – Şirketlerin veya kuruluşların uygulamalarını özel yapan ve kurumlara has özellikler veren uygulamalardır. Orta seviyede kalıcıdırlar ancak değişen iş ve müşteri koşullarına göre sık sık yenilemelere tabi tutulurlar.

Yenilikçi sistemler (System of Inovation) – Gerek hali hazırda işlerin yeniden yapılandırılması konusunda, gerek ise yeni iş kollarına fırsat yakalamak amacı ile yeni girişlerin yapılması için oluşturulmuş birimleridir. Genellikle dış kaynak kullanımı ile ilerlerler ve kısa ömürlü yazılımlar olarak nitelendirilirler.

Eski sistemlerin DevOps disiplinine taşınması her zaman faydalı veya kolay olmamaktadır. Örnek olarak fazla müşteri sayısına sahip ve anlık işlem önemi içeren bazı şirketler adına her müşteri için bir yazılımcı atamak imkansızdır. Bu nedenle bu sistemlerin tek başına Çevik stratejisine geçmesinden ziyade geçişler kademeli olarak gerçekleştirmelidir. Yukarıda üç temel bölüme ayrılmış bir sistem yapısından hareket edecek olursak, Çevik modeline geçişi en kolay ve bu geçişte en az riske sahip sistemler yenilikçi sistemlerdir. Büyük bir sistemi DevOps disiplinine sokmak ve Çevik bir model ile oluşturmak yerine başarısızlık karşısındaki riskimizin en düşük olduğu sistemden başlamak, sistem değişiminde bahsedilen çoğu negatif yanların ortadan kalkmasını sağlayacaktır.

IBM'in danışmanlığını üstlenmiş bir yazılım geliştirici olan Bill Dickenson eski uygulamaların yeni sistemlere ve proje yönetim anlayışlarına dönüşmesinde karşılaşılabilecek beş önemli etmenden bahsetmiştir.

1. Kalite – Kodun ve koda bağlı olarak uygulamanın kalitesindeki yaşanan düşüşler uygulamayı güvenlik ve performans açısından zayıf bırakabilmektedir.
2. Hata yoğunluğu – Uygulamanın kod yapısındaki hataların tüm kodlardaki oranını ifade eder. Genel olarak eski sistemlerde bu tarz durumlarla karşılaşmak daha olasıdır.
3. Karmaşıklık – bir uygulamanın kod kısmındaki karmaşıklık uygulamanın anlaşılmasını veya kaynaklanan hataların saptanmasını zorlaştırmaktadır. Bununla birlikte aynı zamanda uygulamanın farklı yeniliklere açıklığını düşürmekte ve değişikliklere kapalı hale getirebilmektedir.

4. Performans – Uygulamalarda kötü yapılandırılmış sistemsel olmayan döngüler ve aktarım mekanizmaları uygulamaların hızını ve genel anlamdaki performansını yavaşlatır.
5. Güvenlik – hatalı veya zayıf bir şekilde oluşturulmuş kodlar ile ortaya çıkan uygulamalar, normallere göre dışarıdan gelecek tehditlere karşı daha açık ve savunmasızdır.

Burada tanımlanmış risk etmenleri ilk bakıldığında değişimin sahip olduğu riskler gibi gözükmesine rağmen aslında DevOps geçişi yapmak isteyen kurumlar için adeta yoldaki şerit görevini görmektedir. Kurulumumuzu ve geçişlerimizi bu onay listelerine göre kontrol ettiğimiz sürece sonucun riski en aza indirgenecektir.

Eski sistemlerden DevOps sistemlerine geçişte karşılaşılan en belirgin zorluklardan birisi de dokümantasyon eksikliğidir. Özetle belirtmek gerekirse bir sistemde geçmiş zamanda yapılan her işlem kayıt altına alınarak saklanmalıdır. Bu durum hem sistemlerde sonraki zamanlarda yapılacak değişiklikler için kolaylık sağlar, hem de personel değişikliklerinde yeni personelin sistem durumunu anlamasını kolaylaştırır. Ancak eski sistemler bu durumları öngöremediklerinden dolayı disiplinli olarak belgelenmemiş olabilirler. Bu durum bazen sistemlerin gelecekte kullanılmayacağı düşüncesi ile meydana gelmektedir. Bunun dışında belgelenmedeki uyuşmayan disiplinlerde bu duruma neden olabilmektedir. Başlangıçta takip edilen belgeleme yöntemi yeni sistemler için çalışmayabilir veya etkisini yitirmiş bir yöntem takip edilmiş olabilir. Ayrıca belgelemelerin anlaşılırlığı da eski personel ile birlikte zayıflamış olabilir. Belgelendirmenin eksik olduğu sistemler için ortaya çıkan zorluklardan en önemlisi testlerin yapımında karşılaşılan zorluktur. Otomatikleştirilmiş testler DevOps ve Çevik sistemlerinin temelini oluşturmaktadır.

Ayrıca belgelendirilmenin eksik oluşu eski sistemlerde testlerin otomatikleştirilmesini zorlaştırmaktadır. Dokümanız bir şekilde yapılacak testler karmaşıklığı ve test sonuçlarındaki yanılgıyı arttırır. Bu durumu gelişme

gözlemindeki zorluklar, güncelleme konusunda eksik kalan bilgiler, bağlı bulunan diğer eski yapılı sistemlerin etkilenmesi ve belirli analizleri yapmakta gereken zamanın artışı gibi sorunlar takip eder.

5.4.4. Kurumsal kültürde gereken değişiklikler

Sistem ve kuruluşların karşılaştığı birçok zorluklardan, bahsedilmesi ve üstünde durulması gerekenlerden birisi DevOps'un getirdiği ve gerektirdiği kültür değişikliğidir. Bunun temel sebeplerinden birisi insanların geçmiş tecrübeleri, alışkanlıkları ve önyargılarıdır. Şelale sisteminin doğruluğunu kabul etmiş ve senelerini bu şekilde gelişime harcamış insanların yeni ve tamamen farklı bir sistem kullanarak uygulama geliştirme işlemleri DevOps'un sahip olduğu dezavantajları yazılımcıların gözünde büyütür ve sistemi kullanışsız olarak görmelerine neden olur (Saran, 2015).

Şelale modelden Çevik modele geçişte veya diğer bir söylem ile eski bir sistemi DevOps disiplinine entegre etmede kuruluşların yaşadığı en büyük sorunlardan birisi bu geçiş için gerekli olan organizasyon kültüründeki değişime uyum sağlamaktır. Bir ekibin çalışma kültürü olarak Çevik olabilmesi adına öncelikle daha önce sahip olduğu hiyerarşik yapı ile birlikte birimler arasındaki duvarları da kaldırması gerekliliğidir. Özellikle operasyon ve gelişim arasında olan bağımsızlıklar ortadan kaldırılarak iki takımın ortaklaşa görüşleri ile çalışması gerekmektedir. Aksi halde, yani operasyonun yalnızca kod yazma ile sınırlı kalması durumunda organizasyon Şelale modelinde kalacaktır (Saran, 2015).

DevOps yazılım metodolojisinin proje yönetim disiplini olarak entegre süreci ayrıca çalışanların nasıl değerlendirilip ödüllendirilmesi konusunda yapılması gereken bazı yenilikleri gerektirir. Yazılan kod miktarı ve dağıtım ardından kodun hatasız şekilde hizmet etmesi gibi eski değerlendirme kriterlerinin yerine, iyi bir şekilde geliştirilmiş

ve test edilmiş olan kodun temel proje hedefine odaklanmasına ve hedeflerdeki istekleri karşılamaması sonucunda yeniden geliştirme adına sorumlulukların eşit şekilde paylaşılmasına bağlı olarak yeniden oluşturulması gerekmektedir. Geleneksel Şelale sistemlerinde yazılım geliştiriciler geliştirdikleri yazılım için ödüllendirilirken operasyon ekipleri yazılan kodun istikrarını korumakta gerçekleştirdikleri başarılar ile ödüllendirilirler. Bu durum bir nevi ödüllendirmelerin yanlış tanımlanması sorununu ortaya çıkarır. Farklı bir ifade ile aynı sonucu oluşturan iki farklı birim farklı kıstaslar ile ödüllendirilmektedir. Bu durumun temel sebebi eski yönetim bakış açısında iki birimin farklı işlerden sorumlu olması olarak gözükmektedir. Geliştirme ekibi yalnızca kod oluşturma ve uygulamayı serbest bırakma ile sorumluyken, operasyon ekibi sistemsel sorunları gözetleme ve olası durgunlukları ortadan kaldırma ile sorumludurlar. Tek bir takımı veya her iki takımı ayrı ayrı ödüllendirme sistemleri yerine DevOps'un ve Çevik modelin ortaya koyduğu kültür, iki takımın başarıdan önce teşvik amaçlı hedeflere odaklanmasına yönelik çalışmalar ortaya koymasını sunmaktadır. Bu teşvikler, bizzat teşviki sağlayan takımlar vasıtasıyla da kurulabilir. Bu takımlar, iki ekip arası güven inşa etmek, iletişimi geliştirmek ve iş birliğini arttırmak gibi görevleri üstlenerek yalnızca ödül sistemi ile çalışmayıp sistemin kültürel olarak çalışmasına katkı sağlamalıdırlar (Novak, 2014).

Kültürdeki bu değişim yalnızca proje ekiplerine değil, aynı zamanda yöneticilere de belirli zorluklar getirebilir. Eskisi gibi hiyerarşik olan bir yapıda övgünün ve suçlamanın hedefleri belirliyken yeni DevOps sistemlerinde artık proje ekibi olarak tüm bilişim ekibi durum hakkında sorumlu tutulmaktadır. Bu nedenle bireysel ödüllendirme ve cezalandırmalardan ziyade hatanın kaynağını bulmaya odaklı çalışmaların artması gerekmektedir.

İş ve proje tamamlamada eski kuruluşlar tarafında yıllarca benimsenmiş kültürlerin değişimi anlatıldığı kadar kolay olmamaktadır. Bu nedene bağlı olmakla birlikte bugüne kadar DevOps kültürünü kurumsal yapısına uyarlamaya çalışan çoğu kuruluş başarısızlıklar ile karşılaşmıştır. Kaynaklanan sorun bu durum ile sınırlı kalmamakta, kuruluşlar karşılaştıkları bu başarısızlıkların suçunu kendi yapılarındaki değişiklik

stratejilerinden ziyade DevOps metodolojilerine atarak sorumluluğu üstlenmemektedirler. Bu da yapılan hataların saptanmasında kuruluşlara ayrıca bir başarısızlık durumu oluşturmaktadır. Bu hata durumlarına kaynak oluşturan temel neden kurumlardaki yöneticilerin kontrolü her zaman elde tutmak ve projenin kontrol edilebilirliğinin her zaman ön görülebilir olmasına duyulan istektir. Ancak DevOps proje metodolojisi bunun aksine, bilişim ekibine duyulan güvene dayalı çalışmakta olan bir sistemdir.

Hata payının yüksek olduğu geçiş süreçlerine rağmen IBM, HP ve NASA gibi eski sisteme sahip büyük kuruluşlar ise bahsedilen geçişi başarı ile tamamlamış kuruluşlara örnek olarak gösterilebilir. Bunun sonucu olarak bu kuruluşlar, daha kısa yazılım geliştirme süreçleri, iyileştirilmiş ve sonuca daha yüksek başarı yüzdesi ile ulaşılabilen sistemlere sahip oldular. Bu kuruluşların DevOps metodolojilerine başarılı bir şekilde ulaşmalarının altında yatan temel etken kurumsal olarak geçmiş disiplinlerini değiştirmeye açık ve hatalarını kabul etmeye hazır olmalarıdır. Bu kuruluşların dönüşüm sürecinde kendilerini başarıya götüren ortak davranışları şu şekilde sıralanabilir;

1. Gerek teknik gerek genel yönetim kadrolarının her bir biriminin kültürel değişikliği desteklemesi.
2. Teknik ve genel yönetici kadrolarının tüm liderleri bu geçiş sürecine öncülük etmeleri.
3. Yönetim ekibinin tüm bireyleri değişime katkı sağlayan teknik ekip birimlerini ödüllendirmesi ve değişmekte kusurları olan ekip üyelerinin kusurlarını nazik bir şekilde düzeltmek ile ilgili çaba sarf etmesi.
4. Yönetim ekiplerindeki liderlerin, DevOps ekiplerindeki bilişim personeline gerektiğinde yazılım geliştirme sürecinde ve kültürel değişiklik sürecinde liderlik etmesi için izin vermesi (Edler ve ark., 2014).

Gartner Araştırma direktörü Ian Head yapılan araştırmalar sonucu 2018'de DevOps'un çalışma kültüründe sahip olduğu temelleri dikkate almadan uygulamaya çalışan kurumların %90 gibi büyük bir oranla projelerini başarısızlıkla sonuçlandırdığının tahmininde bulunmuştur. Hata oranı bu kadar yüksek olan bu tahmine zemin hazırlayan temel gerekçe, yönetimin DevOps'a geçmek istese bile bunu bilişim personeline kabul ettirememesi ve bu sebeple oluşan davranış farklılıklarıdır. Buna ek olarak diğer hata sebepleri merkezi bir DevOps yönetimi oluşturamamak, ilgili birimlerin iş yükümlülüklerini anlayamamak ve doğru şekilde sorumlulukları paylaştıramamak, ulaşılmaz hedefler belirlemek ve DevOps entegrasyonunu Şelale sistemlerine hatalı metotlar kullanarak entegre etmeye çalışmalarıdır. Bu konu farklı bir kaynakta Gruver ve Mauser tarafından da benzer şekilde savunulmuştur. Gruver ve Mauser'e göre yönetimlerin geleneksel ve merkezi sistemlerini değiştirmeleri gerekir. Aksi takdirde başarısızlık oranındaki artış kaçınılmaz olur. Geçmişte olduğu gibi aynı hataları yapmaya devam ederler ve gelişim için gereken esnekliği ortadan kaldırmış olurlar (Gruver ve Mausser, 2015; Saran, 2015).

Bu örneklerden yola çıkarak söylenebilir ki, DevOps bir kültür değişimini temsil eder. Gerek gelişimi yazan mühendislerin kendilerini operasyona daha fazla adapte etmeleri, gerekse operasyon ekimlerinin meslektaşlarının yazdığı kodlara daha duyarlı ve empati ile yaklaşması bu kültürün temel değerleridir. İlgili yönetimin de bu bağı kurmada ve empatiyi arttırmada istekli olması gerekir. Özellikle yönetim ekibi, oluşabilecek kültür çatışmalarının önüne geçmelidirler.

5.4.5. Eski sistem mimarisindeki zorluklar

Büyük sistemlerin DevOps'a entegresi küçük ölçekli kurumlara göre daha zorludur. Bu aşamada zorluğu oluşturan temel durum, kurumlarda oluşan geniş ölçekli ve zaman içerisinde karmaşıklaşmış olan uygulama yapılanmasıdır. Diğer bir söylem ile eski sistem mimarilerinde sistemlerin kullanım süresi boyunca yer almış olduğu uzun zaman alanı içerisinde yamalar ile düzenlenmiş ve iç içe geçmiş karmaşık yapısıdır. Bu durumda, geçişlerden önce gerçekleşmesi gereken kod ayrıştırma, çalışma zamanı analizi ve uygulama içi bağımlılıkların haritalandırılması, geçiş konusunda ayrıca bir öneme sahiptir. Ayrıca kurumsal karmaşa olarak adlandırılabileceğimiz kurumlardaki çalışan binlerce Bilişim teknolojileri personeli ve projelerde görev alan bu personellerin birçok farklı birime bölünmesi, DevOps geçişini daha karmaşık hale getirmektedir. İş kolunda bulunan her bir farklı alan, ki bu alanların zaman içerisinde kökleşmiş olduğunu ve diğer iş kollarından tamamen farklılaşmış olduğunu varsayarsak, bu durumu daha da zorlaştırmaktadır. Bu fazla sayıda ve komplike yapıda iş kollarının her birinin bölünmesi ve kısa süreli geri dönüşlerin sağlanması, bu tarz yapılanmalarda Çevik model için durumu zorlaştırmaktadır.

Bahsedilen bu sorunları aşmak amacı ile önerilen bir çözüm yolu, DevOps geçişlerinde ani bir geçişten ziyade bir denge sistemi ile geçiştir. Daha önceki eski sistemlerin mimari yapısında belirtildiği üzere, eski sistemleri önem sırasına göre birkaç farklı sistemlere böldüğümüzde ve geçişi en az riskliden başlatarak gerçekleştirdiğimizde gerek kurumsal geçiş adına gereken tecrübe, gerek sonradan ortaya çıkan risklerin gözlemlenmesi sağlanmış olur. Yani eski ve büyük yapılı sistem mimarilerini tamamen değiştirmek yerine, önerilen çözüm yolu bu sistemleri kısmen ve uygun olan en kolay şekilde dönüştürmektir. Böylece, bahsedilen karmaşık yapıdaki sistemler DevOps için uygun değilmiş gibi gözükse de veya sahip olduğu yapı gereği dönüşümün normale göre zaman ve mali maliyetlerinin yüksek olması durumu göz önünde olsa da durum kuruluşların bu dönüşümü gerçekleştirmemesi gerektiği anlamına gelmemektedir. Bu durum yalnızca yönetimin ve teknik liderlerin farklı birtakım adımları takip ederek süreci gerçekleştirmesini

gerektirir. Ayrıca kuruluşun yapısına göre Çevik modelin herhangi alt bir disiplini seçilebilir. Sonuç olarak görülebilir ki, en karmaşık yapıdaki mimariler bile kültür değişiminin ardından bilişim ve yönetim ekibinin ortak çalışmaları ile esnek bir yapıya büründürülebilir (Gruver ve Mouser, 2015).

5.5.Eski Yapılı Sistemlerin DevOps Geçişlerine Örnekler

5.5.1. Örnek 1: NASA'nın DevOps disiplinine geçişi

NASA, DevOps disiplinini Çevik model ile ilk olarak 2013 yılında görev kontrol yazılımının geliştirilmesi sürecinde uygulamıştır. Bahsedilen bu çalışma NASA'nın AMES'deki tasarım ve geliştirme ekibini, bu ekibin müşterilerini ve NASA'daki görev kontrol ekibini içeriyordu. NASA'nın bu önemli geçişe sebep olarak ortaya koyduğu genel problemler, eski sistemin sahip olduğu uzun proje oluşturma döngüleri ve projenin test aşamasında karşılaşılan herhangi bir hatanın dönüşünün maliyetli olmasıdır. Önceki kullandıkları uygulama geliştirme modellerinde altı aylık gelişim döngülerine sahip olan NASA, bu döngülerin çok uzun olduğunun altını çizerek, yayınladığı makalesinde müşterilerin bu süreyi kabul etmelerine rağmen oluşan herhangi bir hatanın yeni bir altı aylık döngüye tekabül ettiğini vurgulamıştır (Trimble ve ark., 2016).

NASA'nın tasarım ve geliştirme ekiplerinin Şelale modelinde karşılaştığı diğer önemli zorluklar şunlardır;

1. Önceliklerin belirlenmesinde, geliştirilmesinde, sisteme entegresinde ve test süreçlerinde yaşanan zorluklar
2. Müşteri iletişiminin olmaması ve bu nedenle sonuçlarda karşılan anlaşmazlıklar.
3. Geliştirme ekibinin karşılaştıkları sorunları müşteriler ile konuşamaması

4. Proje sürecindeki odağın sonuçta istenen yerine alt sistemlerin ve süreç bazlı olması.

Geçiş aşamasının başlıca planı; çalışma odağındaki her grubun aynı şekilde bir uygulama geliştirme projesine ihtiyaç duyup duymadığı, kurum içi organizasyon yapılarını bozmadan yeni fonksiyonların sisteme eklenmesi ve hangi sisteme hangi fonksiyonları ekleneceği ayrımının yapılmasını içermekteydi (Trible, 2013).

NASA'nın geçiş evresindeki ilk hedefi, sahip oldukları proje geliştirme döngülerini kısaltmaktı. Bu durumu ilk olarak altı haftaya, ardından üç haftaya düşürmeyi başaran NASA bilişim ekipleri, bu üç haftalık döngülerin her birisini bir sprit olarak değerlendirdi. Başarı ile tamamlanmış dört spritin ardından kod dağılımı gerçekleştirilmeye başlandı. Bu da bir bütün olarak başlangıçtan sonra kadar geçen üç aylık kısa bir teslim süresi ve minimuma indirgenmiş hata oranını beraberinde getirmiş oldu. Özellikle 3 haftalık her sprit süreçlerinin sonunda gerçekleşmesi gereken testlerin müşterilerin gözetimi altında gerçekleştirilmesi, proje sonunda karşılaşılabilen memnuniyetsizlik sorununu da ortadan kaldırmış oldu.

Bu başarının ardından NASA'nın açıkladığı bir sonraki hedefi, Görev Operasyon Sistemleri (MOS) olarak adlandırılan, yalnızca yazılımı değil aynı zamanda bütün kurumun sahip olduğu çalışma metodolojisini de bir Çevik modele döndürmektir. Bunun uzun bir süreç olduğunu belirten NASA bilişim ekipleri, yalnızca yazılım olarak yapılan geçişlerin Çevik modelin verimliliğini tam olarak almak adına yeterli olmadığını, geçişin aynı zamanda operasyonlarda da kullanılması gerektiğini ve böylece bir nevi DevOps disiplini ile çalışılması gerektiğini vurgulamaktadırlar.

Sonuç olarak Çevik model SDLC'yi görev kontrol sistemlerine entegre eden NASA bilişim ekipleri, proje kontrol döngülerini 6 aydan 2-3 haftalık sürelere döndürmeyi başardılar. Bu dönüşüm sonrasında her bir döngünün net bir amacı, açıkça

belirlenmiş oldu. Ayrıca bunların yanında Şelale modelindeki ağır işleyen ve verimliliği düşüren dokümantasyon süreci daha uygun ve kolay işlenen bir dokümantasyon süreci ile değiştirilmiş oldu. Ek olarak, uygulamanın gelişimini izlemek ve oluşan hatalara daha hızlı müdahale edebilmek adına daha net bir stratejik yol haritası belirlenmiş oldu Burada görülen örnek, NASA gibi büyük ve eski sistemlere sahip bir kurumun sahip olduğu zorluklara rağmen Çevik ve DevOps disiplinlerine geçişlerini başarmalarıdır. Bu başarıda izlenen yöntem, sistemin tümünün geçişinden ziyade, geçişin en makul olan noktalara uyarlanmasıdır. Daha sonrasında sağlanacak uyumlar ile diğer daha riskli kurumsal birimlere bu geçişin yapılması, NASA'nın belirttiği bundan sonraki hedefleridir (Trimble, 2013).

5.5.2. Örnek 2: IBM'in DevOps disiplinine geçişi

IBM ve IBM'in sahip olduğu müşterileri ile birlikte DevOps disiplinine geçişinde aktif rol alan Rosalinda Radcliffe, bu geçiş sürecinde karşılaştıkları zorluk ve sonuç olarak elde ettikleri başarıları detaylı bir şekilde belirtmiştir. Geçişten önce bu geçişe neden olan en önemli durumun eski proje sistemlerinde karşılaştıkları en az 18 aylık proje yaşam döngüleri olduğunu vurgulayan Radcliffe, geçiş sonunda ulaşmak istedikleri hedefleri; müşterilere daha kaliteli hizmet sunabilmek, müşterilerin gerçek anlamda birbirlerine entegre olmuş yapılara sahipliğinden emin olmak, Müşterilerin yeni sistemlere adaptasyonlarını kolaylaştırmak ve müşteriler için oluşturulan projelerin daha hızlı ve daha verimli bir şekilde teslim edilmesi olarak sıralamaktadır. Kısaca bir kazan kazan stratejisi neticesinde çıkan bu karar hem müşterileri hem de kendi sundukları hizmetleri verimli bir hale sokmayı amaçlamıştır (Gajda, 2018).

IBM'in gerçekleştirmiş olduğu geçiş sürecinde izledikleri stratejilerden bahsetmek gerekirse, Radcliffe'in tanımlamalarına göre Çevik modele geçişten önce genel anlamda eski uygulama geliştirme modelleri ile yürütülen 17 ayrı üründen bahsetmektedir. Bazı durumlarda bu ürünlerin birbirlerine bağları bulunurken bazı

durumlarda ise uygulamaların tamamen birbirlerinden ayrı oldukları vurgulanmaktadır. Bunun üzerine bahsedilen bu ürünler ile ilgili her şeyi belirli müşteriler hariç durdurma kararı aldıklarını ve geçişi bir teslim hattı kurulması üzerine belirleyerek projelendirdiklerini belirtmektedir. Geçiş stratejisindeki asıl amaç müşterilere teslim edilen bu 17 farklı ürünün birbirleri ile aynı sisteme sahip olmalarını sağlayarak, yapılacak herhangi bir güncellemede uygulamalar arasında fark kalmadan bu geçişin sağlanması hedeflenmiştir (Gajda, 2018).

Geçiş süreci yaklaşık 4 ay sürdüğünü belirten Radscliffe, geçişin tüm ürünler için aynı anda gerçekleşmediğini, belirli bazlara sahip ürünlerin önceliği ile başlatılan bir geçiş süreci ile başlandığını ifade etmektedir. Geçişi tamamlanan her ürünün müşteri tarafından talebinde, gelişimin müşteriler ile birlikte daha sık yapılan proje takip döngüleri ve daha çabuk teslim edilen projeler ile sonuçlandığı ayrıca bildirilmektedir. Bu durumu bir nevi üretim hattı metodolojisine dönüşüm olarak özetleyen Radscliffe, sonuç olarak ulaşılan durumun her ne ürün talep edilmiş olursa olsun sistemin DevOps disiplini ile çalışarak uygulama teslim etmeye başladığını ifade etmektedir.

Radscliffe e göre, strateji ve geçiş sonrasındaki uygulama teslim süreci sonunda her bir ürün için tek bir kaynak oluşturulmuş, her ürün için görevler otomatik hale getirilmiş, özellikle test süreçlerinin otomatikleştirilmesi de gerçekleştirilmiştir. Bu da geceleri zamanın test süreçleri ile harcanmasını ve böylece ayrıca bir zaman ve mali tasarrufun oluşturulmasını sağlamıştır. Bu zaman kısalığı ve kazanılan hız, sonuç olarak müşterilere daha hızlı ve daha talebe uyan hizmetlerin sağlanması olarak gerçekleşmiştir (Gajda, 2018).

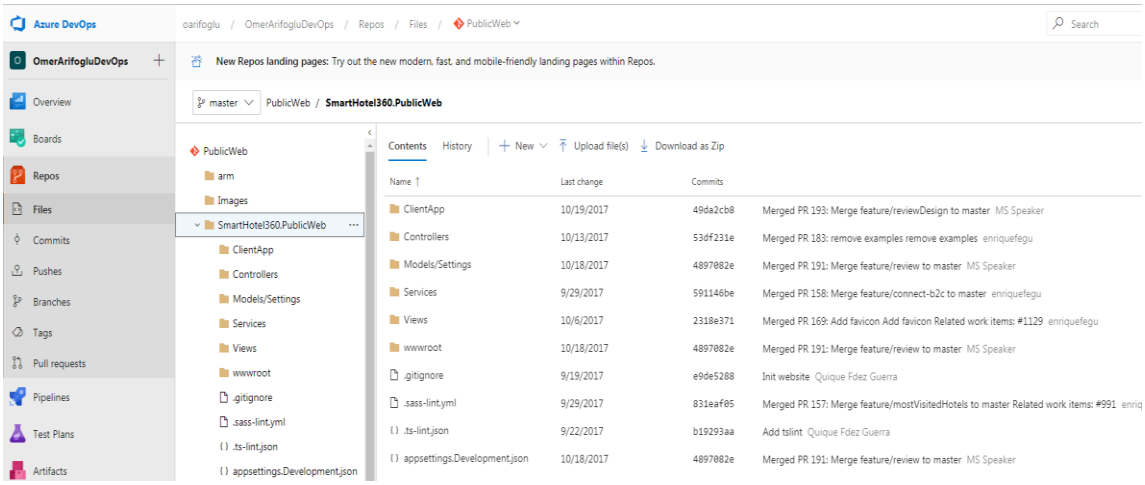
Sürecin katkılarına ek olarak süreç boyunca neler öğrendiklerini ve bu süreci daha sonra takip edeceklere tavsiyeleri olarak Radscliffe, öncelikle geçiş aşamasındaki ekibe verilmesi gereken zaman ve eğitimin önemini vurgulamaktadır. Kullandıkları sistemin tamamen değişmesi ile birlikte eskisi kadar verimli çalışamayan ekipler, bu

geçişin başlangıçta bir hayal kırıklığı yaratması düşüncesini oluştursa bile, bunun geçici bir durum olduğunu ve bu durumun eksik bilgilerin doğru eğitimler ile tamamlanması sayesinde aşılabileceğini vurgulamıştır. Bunun yanında bilişim ekiplerine bu öğrenme aşamasında verilmesi gereken zamanın da önemini ayrıca vurgulamıştır. Her ekibin ayrı ayrı çalışması kültürünün bozumundan bahseden IBM geliştirme ekibi üyesi, farklı takımlar ile aynı ürünü çıkarmak adına birlikte çalışmanın öğrenilmesi gerektiğini de vurgulamaktadır. Önceki paragraflarda belirtildiği gibi ilk zamanlarda yaşanan verim düşüklüğü yalnızca ekip olarak değil, yönetici ekibi olarak da anlayış ile karşılanmalıdır. Bunun önemini ayrıca vurgulayan Radscliffe, yönetimin desteği olmadan böyle bir geçişin mümkün olmadığını belirtmiştir.

5.5.3. Örnek 3: Azure DevOps Kullanılarak Oluşturulmuş Örnek Uygulama

Tezimizin bu kısmında, gelişim süreci tamamlanmış bir uygulamanın operasyon kısmının otomatikleştirilmesi uygulama örneği olarak kullanılmıştır. Bu otomasyonu gerçekleştirmek adına kullandığımız uygulama piyasada sıkça kullanılan Microsoft'un Azure DevOps aracıdır. Gerçekleştirilen otomatikleştirmeleri tanımlamaktan önce örnekteki uygulamanın .Net Core teknolojisiyle geliştirildiğini, ilişkisel veri tabanı olarak MSSQL kullanıldığını, canlı sisteme aktarılmadan birim testlerinin çalıştırılma ön koşulu olduğunu ve bu testler başarılı olduğunda uygulamanın test ortamına geçtiğini varsayacağız. Daha ayrıntılı anlatmak gerekirse, varsayımlarımızda bahsi geçen test sunucularına uygulamanın dağıtılması, proje kapsamında Azure kullanılarak otomatize edilmek isteniyor. Bunun ardından uygulamanın test ortamına başarıyla çıktığında sorumlu kişiye e-mail atıp sürecin bir raporunu iletmesi bekleniyor.

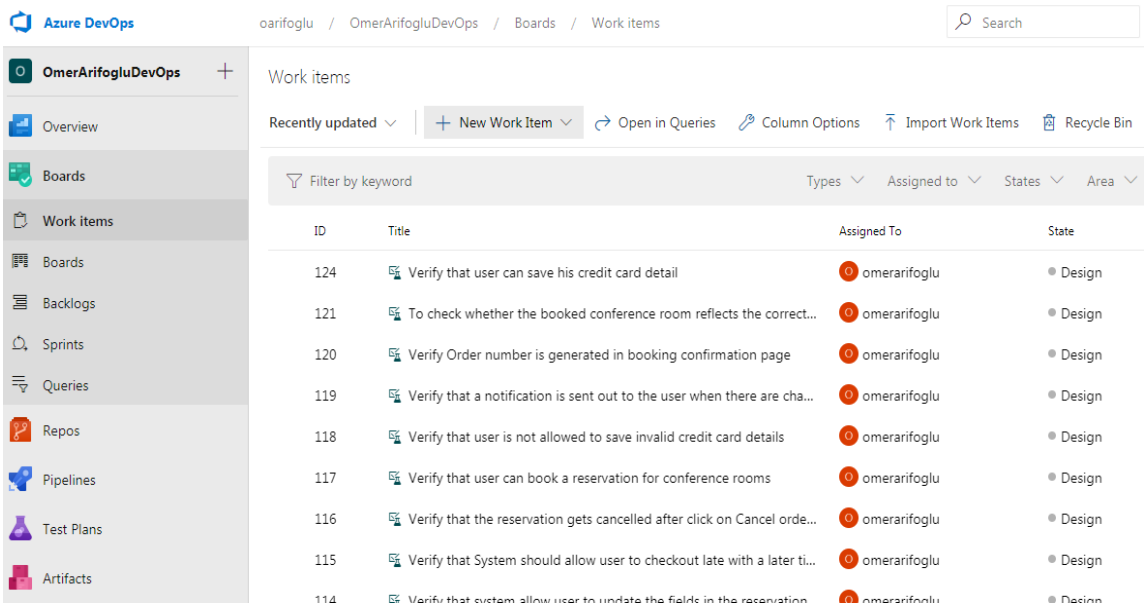
Bu hedefler doğrultusunda gereken kod geliştirildikten sonra kod 'git' alt yapısıyla saklanıyor, Şekil 5.1.'de görülebileceği üzere git hesabı Azure DevOps aracıyla ilişkilendirilebiliyor.



Şekil 5.1. Git Repository Konfigurasyonu

Bunun uygulamaya kattığı en önemli yetenek, yazılım geliştiricilerinin kendi bilgisayarlarında ‘build’ komutuyla süreci yürütebilmelerini ve henüz geliştirme aşamasında bile kodun gereken testleri geçip geçmediğinin kontrol edilmesi ardından ilgili raporun çıkartılabilmesini sağlamasıdır.

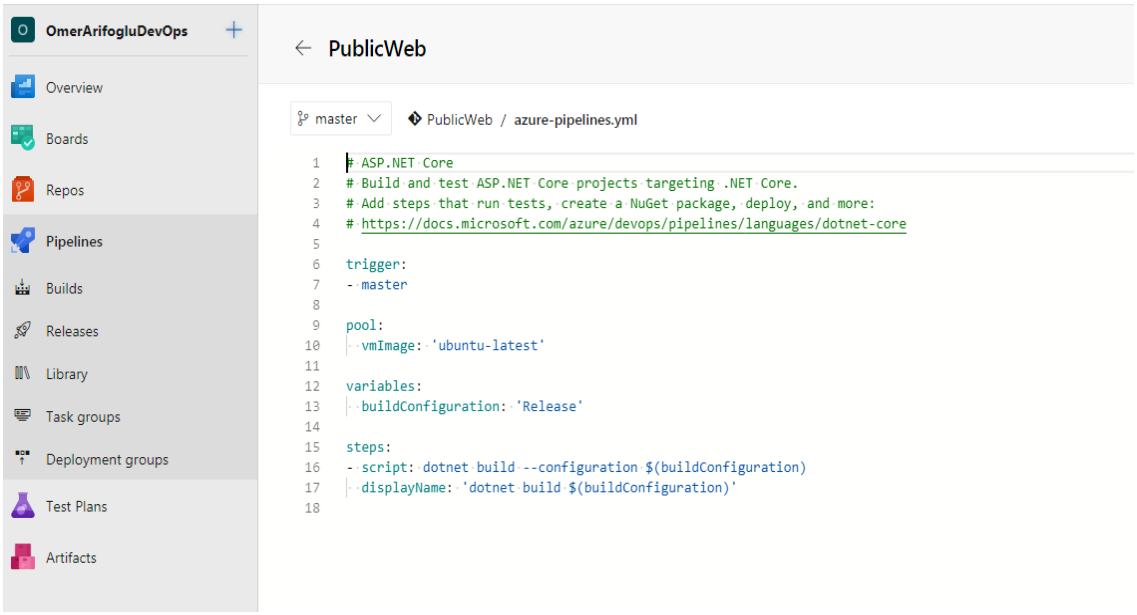
Projenin geliştirme maddeleri Agile metodolojilerine veya Kanban metodolojilerine uygun olarak geliştirmek isteniyorsa, örnekte kullanılmış olan araç bu metodolojileri de Şekil 5.2.’de görülebileceği gibi sisteme entegre etmeye olanak tanımaktadır.



Şekil 5.2. Work Item Sayfası

Uygulama aynı zamanda Ekibe ilgili üyelerini buradan tanımlamayı, iş atamalarını gerçekleştirmeyi, atanan işlerin durumlarını ve atanmış olan işlerin durumlarını takip edebilmeyi sağlamaktadır.

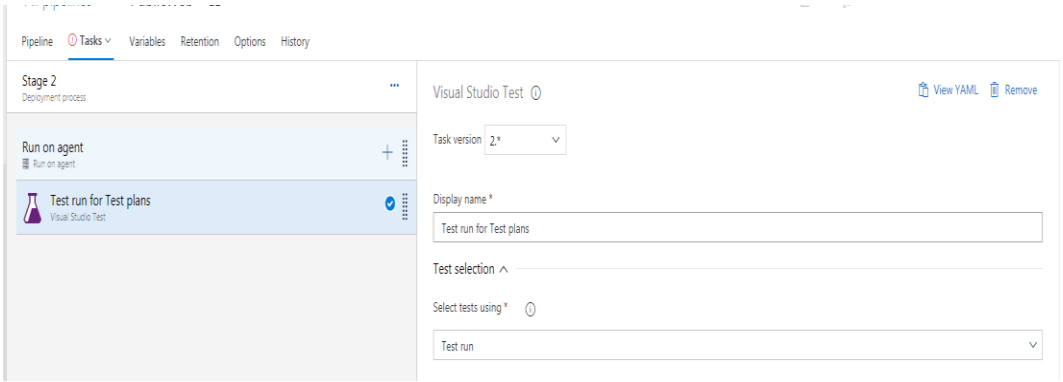
Bu otomasyona geçmeden önce geliştirilen kodun build adımının tanımlaması yapılmalıdır. İlgili adımda hangi teknolojiyle build'in nasıl gerçekleştirileceği önemlidir. Komutlar, farklı dillere ve framework'lere göre değişim gerektirmektedir. Örnek olarak, Şekil 5.3.'te görülebileceği üzere, repository'si bağlı olan uygulama .Net Core altyapısıyla geliştirildiğinden dolayı, dotnet build komutunu kullanması gerekmektedir. Derleyicideki build komutu, Şekil 5.3.'te görülen yml uzantılı olan dosyayı çalıştırır ve süreci buna göre ilerletmeye devam eder.



Şekil 5.3. Build Konfigurasyon Sayfası

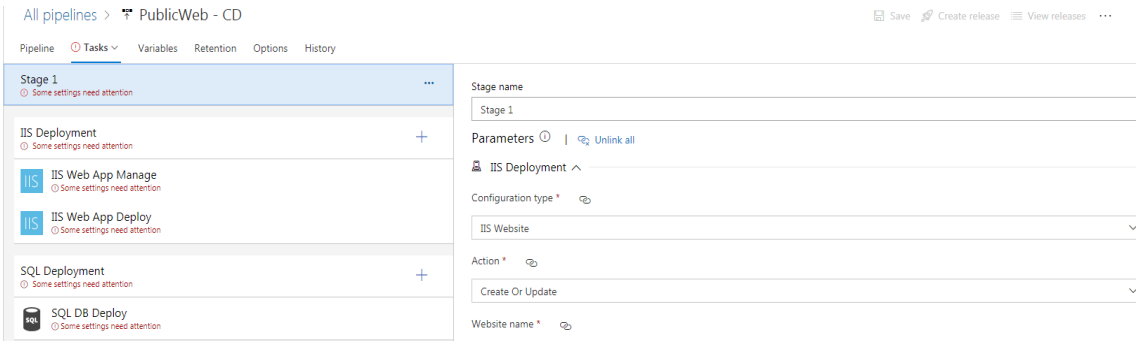
Kurguda olduğu gibi eğer build adımı doğruysa ve syntax hatası olmayan bir kodla çalışılıyorsa, test adımına gönderilecek ve sonuçlarına bağlı olarak sürece devam edecektir.

Bu uygulamada bir önceki adımları izlemek koşulu ile farklı test adımları eklenebilmektedir. Şekil 5.4. DevOps Pipeline'ına görülebileceği gibi bir test adımı daha eklenmenin örneğini göstermektedir.



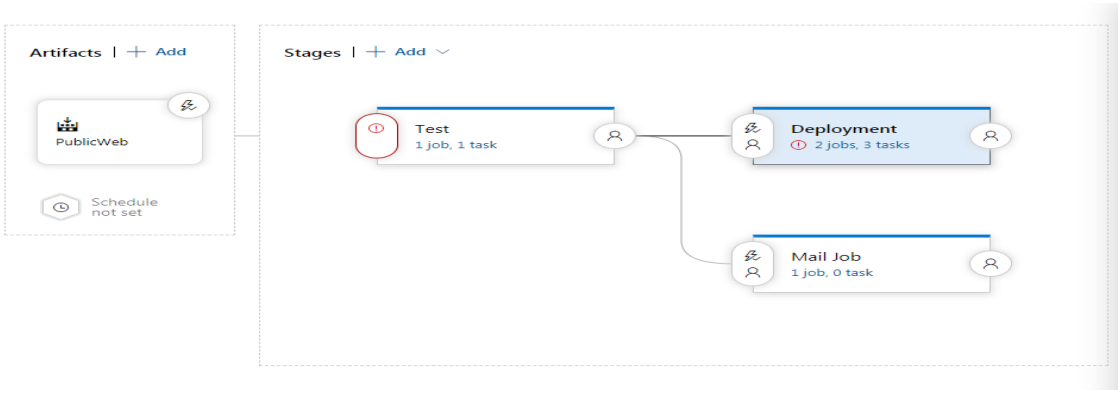
Şekil 5.4.-Test Adımı Ekleme

Bu aşamanın sonraki sürecinde testlerin başarılı olması durumunda Pipeline'a bir deployment adımı eklenir. Bu şekilde örneğe sadık kalarak, IIS uygulama sunucusu arkasında çalışacak, deployment adımında paketlenerek sunucuda çalıştırılacaktır. Şekil 5.5.'te görülebileceği üzere bu geçiş SQL Script'i içeriyorsa SQL Deployment adımları da konfigürasyonları yapılarak bu adıma eklenmelidir.



Şekil 5.5. Web Server Konfigurasyonu

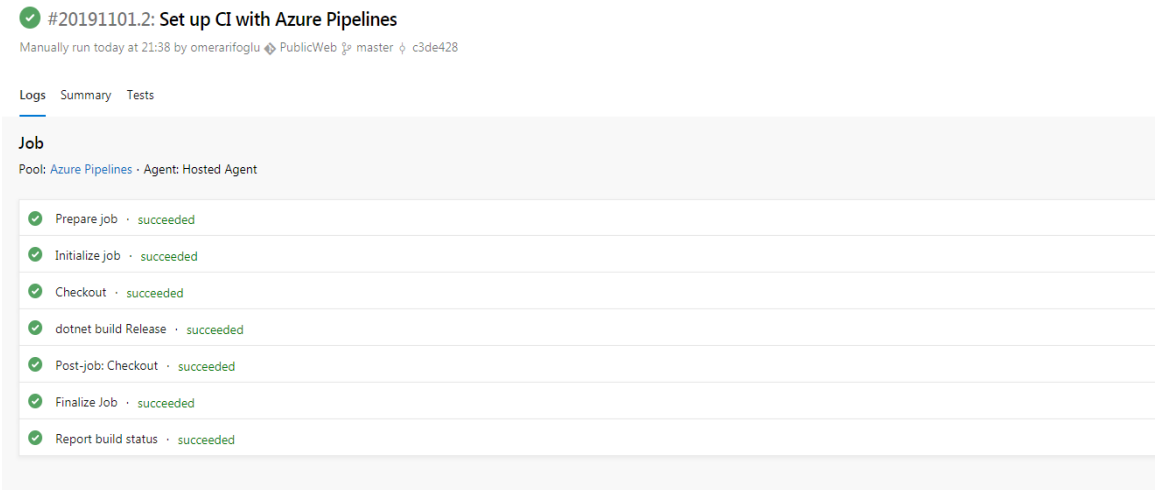
Bu adıma kadar herhangi bir sorun olmadan gelen uygulamaya bir de e-mail gönderme task'ının eklenmesi gerekir. Bu task'ta eklendikteon sonra nihai Pipeline modeli Şekil 5.6.'daki gibi olacaktır.



Şekil 5.6. Pipeline Diagram

Sonuç olarak ulaşılan yapı, kurgulanmış olan adımlarla birlikte herhangi bir son kişi müdahalesine ihtiyaç duymadan, uygulamanın build olması durumunda Şekil

5.7.'deki gibi gerekli log takibi yapılabilme yeteneğini kazandırmaktadır. Aynı zamanda sonuçların her adımı raporlanabilir bir şekilde elimize geçmektedir.



Şekil 5.7. Süreç Takip Sayfası

Geliştirilen örnek haricinde, benzer adımlar izlenerek uygulamaya ihtiyaç duyulduğu miktarda adım eklenebilmekte, farklı sunuculara deployment işlemi gerçekleştirilebilmekte, farklı yazılım dilleri derlenebilmekte ve testleri çalıştırılabilmektedir. Bahsedilen farklı altyapılarla oluşturulmuş olan projelerin hepsi tek bir çatı altında bir pipeline içerisinde işletilerek gereken dağıtım yapılabilir. Büyük ölçekli uygulama geliştiren kurumlara sağladığı avantajlar ise tez süresince tanımlandığı gibi esneklik, takip edilebilirlik, şeffaflık ve ölçülebilirliktir.

5.5.4. Örnek 4: DevOps Geçişi Verimlilik Ölçümü Uygulamalı Örneği

Bu örnek vaka çalışmasında hedeflenen bulgu, DevOps geçişleri sonrası verimlilik artışını uzun ve kısa dönemlerde olmak üzere hangi farklı değişkenler ile ölçümlenebileceğini göstermektir. Örnek vaka çalışması, küçük ve orta ölçekli işletmeler için şirket içi geliştirilen bulut tabanlı bir yazılım ürün paketi aracılığıyla hizmet sunan Finans sektöründe bir yazılım şirketidir. Ürünleri, bir hizmet modeli

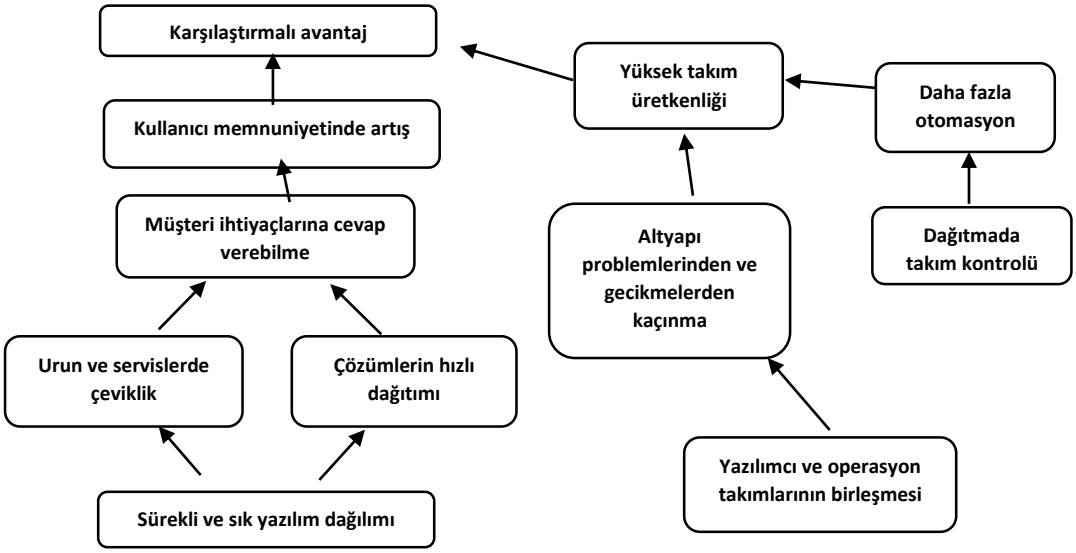
olarak yazılıma dayanmakta ve abonelikle satılmaktadır. Yapılan çalışma, genel olarak geçiş sürecindeki sorumlular tarafından gerçekleştirilen görüşmelerin verdiği sonuçlar ile tamamlanmıştır. Bunun yanında özellikle yazılım geliştirme ve Operasyon personelinin de görüşlerine, vaka çalışmasında yer verilmiştir.

Öncelikle vaka çalışmasında odaklanılan ilk bölüm, geçiş süreci öncesi kurumun genel yapısı ve kurumu DevOps geçişine iten temel sebepler olacaktır. Bu konuda ilgili yönetim ekibinin DevOps entegresinden önce şirketin genel yapısını ve son ürünü çıkarmak konusunda gerçekleşen adımları tanımlamalarına göre; DevOps'un uygulanmasından önce, şirketin ürün ekibi platform ve ürün geliştirme olmak üzere iki ayrı takımdan oluşuyordu. DevOps'tan önce, şirket geleneksel bir veri merkezinde barındırılan monolitik uygulamasını sürdürüyor ve geliştiriyordu. Bu çalışma modeli, başlangıçta şirkete iyi hizmet verebilmiş ve yazılımın başarısına ilk aşamalarında hızlı bir şekilde katkıda bulunabilmişti, ancak daha sonra ortaya çıkan iş hızının artması ve taleplerdeki karmaşıklıklar çok sayıda eksiklikler gün yüzüne çıkarmaya başladı.

Firma, DevOps öncesi üç sıkıntıyı su şekilde sıralamaktadır. Birinci temel etken değer zincirinin yanlış bölümünde ayrılmasıdır. Ürün geliştirme ekipleri, genellikle iş ağı kurma ve gerekli operasyon değişiklikleri ile birlikte hızlı bir şekilde istenen ürünü gönderme ile sorumludur. Operasyon ekipleri birçok ekipten gelen taleplere hizmet eder ve sıklıkla ürün ekibi zaman çizelgelerini dikkate almadan kendi iç önceliğini belirler. Tanımlanan bu iki ünite arasındaki uyum eksikliği nedeniyle doğal olarak verimsizlik ve hata oluşumu meydana gelir. İkinci temel etken Operasyon ve Ürün ekipleri, teşvik ve kontrol tanımlamaları kapsamında bir uyuşmazlık içinde faaliyet göstermesidir. Geliştirme ekiplerinin sahip olduğu yeteneklere rağmen Operasyon ekipleri vakada bahsedilen kurum içinde onların yerine performans ve çalışma süresinden sorumlulardı. Buna benzer şekilde geliştirme ekipleri büyük bir çeviklik ve hıza sahip dağıtım biriminden sorumluyken, operasyon ekipleri yazılım geliştirme yaşam döngüsünün daha önemli bölümlerinin kontrolünde idi. Buda, yeteneklerin doğru paylaştırılmaması sonucu verimsizliği

ortaya koyuyordu ancak burada, sorumlulukların yeniden dağıtımından elde edilecek kesin ve optimal bir çözüm mevcut değildir. Bu nedenle, iş bölümünün dağıtımından ziyade bir araya getirilerek ortak çalışma haline sunulması vakadaki kurum adına önemli bir çözüm olarak ortaya çıkmaktaydı. Son olarak, organizasyonun yapısında değişen bir kısım teknolojiler ve yöntemler, belirli ekiplerin belirli konularda bilgi sahibi olmasını gerektiriyordu. Kurum, daha önce hali hazırda çevik modelin belirli aşamalarını kendi sistemlerine entegre etmişti. Bunlardan en önemlisi Çevik sistem için gereken test gibi kolaylaştırıcı özelliklerdi. Buna bağlı olarak kodlama anlamında altyapıya izin veren bir host sağlayıcısına geçme gereksinimi duyulmuş ve bu gerçekleştirilmişti. Bu hamle, yazılım geliştirme ekiplerindeki personelin daha kapsamlı bir bilgi birikimine veya ekstra eğitimlere zaman harcamalarını gerektiriyordu. Çünkü bahsedilen geçiş, kurumsal işleyişi karmaşıklaştırmış ve operasyon birimlerin belirli aşamalarını yazılım birimlerine yükler olmuştu.

Vaka çalışmasında kurum içi yönetimin verdiği bilgilere dayanarak elde edilen sorunlara çözüm olacak ve kurumun verimliliğini arttıracak, diğer bir söylem ile DevOps'u benimsemeyi motive edici faydalar, stratejik, taktik ve operasyon ile ilgili sürücüler de dahil olmak üzere Şekil 5.8'de şematik olarak ayrıca gösterilmektedir.

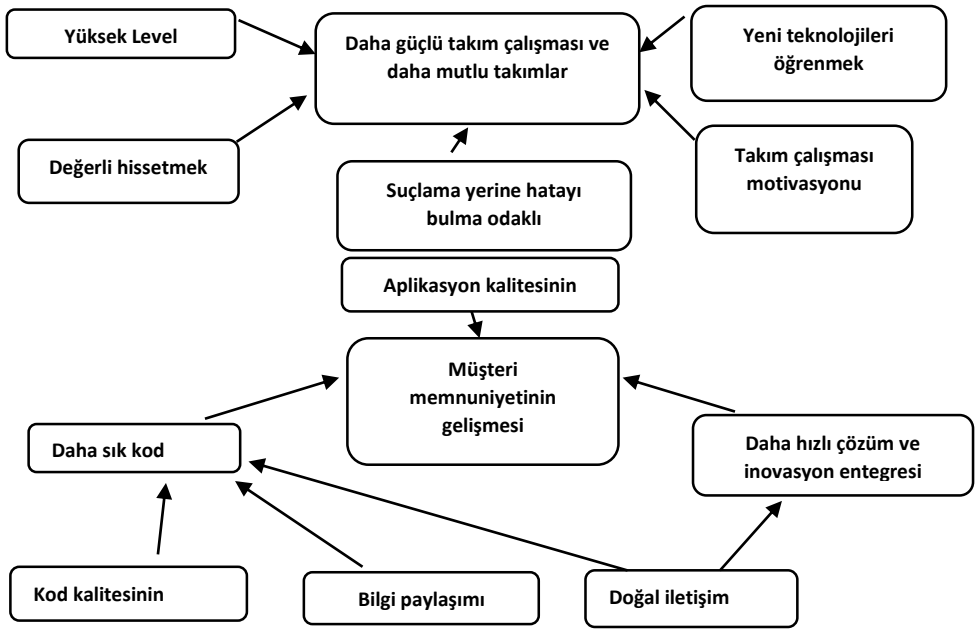


Sekil 5.8. DevOps'u benimsemeyi motive edici faydalar

Sonuç olarak, şirket bir dizi temel değişiklik yaptı. Önceleri, talep üzerine bulut bilişim platformu sağlayan bir sunucuya maliyetli bir geçiş gerçekleştirdiler. Bu değişiklik, ürün ekiplerinin kendi bağımsız altyapılarına erişmelerini ve bakımlarını yapmalarını sağladı ve uçtan uca kontrol sağlayarak ihtiyaç duydukları şeyi tasarlamaları ve inşa etmeleri için mühendislerle daha yakın çalışmalarını sağladı. Bu uygulamanın masrafının büyük bir kısmı, monolitik uygulamalarının büyük bölümlerini bağımsız olarak ölçeklendirilen ve daha önce çalıştıkları Hizmet Seviyesi Anlaşmaları olan bu yeni platform ortamında çalışmak üzere yeniden yazmak için harcandı. Ekip perspektifinden bakıldığında, şirket operasyon ekibinin silolarını dağıtarak ve platform mühendislerini ürün geliştirme ekiplerine taşıyarak "gömülü operasyon modeli" tanıtıldı. Mevcut geliştirme görevlerinin yanı sıra, ürün geliştirme ekipleri daha sonra yeni edindikleri işletme becerileri ile kendi platformlarının operasyonlarından ve maliyetlerinden sorumlu oldu. Odak noktası, uçtan uca kabiliyetli yetenekleri olan ekipler yaratarak ürünü sevk etmek ve kullanmak konusunda teşviklerdi. Bu tür takımların yaratılması, doğru beceri setini edinmeye yatırım yapmayı içeriyordu.

DevOps geçişinin sonrasında tamamlanan bir yılın ardından kurum içi verimi gözlemlemek adına gerek yönetim gerek ise çalışanlar ile gerçekleştirilen görüşmeler sonucunda elde edilen bulgulara göre, DevOps kuruma her an beklenmedik şekilde oluşacak problemleri çözerek günü dolduran bir yapıdan ziyade her türlü probleme hazır, daha sistematik ve disiplinli bir yapı kazandırmıştır. Bu durumu daha ayrıntılı belirtmek gerekirse, ortak iş birliği ile çalışmaya başlayan takımlar, daha mutlu ve birbirlerine daha entegre olmuş şekilde ürün geliştirme sürecini tamamladıkları örnek gösterilebilir. Tetikleyici etken olarak tanımlanmamasına rağmen, bu fayda görüşülen personelde gözle görülen bir sonuçtu.

Şekil 5.9'da gösterildiği gibi, artan takım mutluluğuna ve katılımına katkıda bulunan DevOps ile ilgili başka faydaları da vardır. Yeni DevOps işleyiş tarzında ürün ekipleri kendilerini daha değerli hissederler.



Sekil 5.9. DevOps adaptasyonunun görülen sonuçları

Örnek vermek gerekirse ürün geliştiriciler veya operasyon ekibi üyeleri, karanlıkta kapandıkları odalarında yalnızca rakamlar ve formüller yazarak hayatlarını geçirmek yerine çalışmalarının gerçek müşteriler üzerindeki değerini ve etkisini görebiliyorlardı. Bundan ziyade üretiminde katkı sağladıkları son ürünün ne olduğunu ve bunun karşılığında nasıl bir memnuniyet aldıklarını tecrübe edebiliyorlardı. DevOps, geliştirme ekibinin tüm şirket, ürün ve müşteriler tarafından nasıl kullanıldığı hakkında daha kapsamlı bir görüşe sahip olmasını sağladı. Buda farklı sorunlara farklı çözüm önerileri getirilmesini ortaya çıkardı. Yüzü gülen değil, işi bilen müşteri temsilcilerini proje oluşum aşamalarına dahil ederek, projenin en başından öngörülemeyen hataların engellenmesi sağlandı.

Bunların yanında ekip üyelerinin birçoğu yeni teknolojiler hakkında bilgi sahibi olmaktan net bir şekilde zevk aldılar ve çalışmalarının bir parçası olarak yeni DevOps teknolojisini sağlayanlar hakkında bilgi edinme ihtiyacı ile motive oldular. Burada sağlanan yeni bilgiler, dışarıdan ek bir ödenek ile getirilen kişiler vasıtası ile değil, işin içinde olan takım arkadaşları sayesinde hem ek bir ücret ödemeden hem de takımlar arası iletişimi geliştirerek sağlandı. Ayrıca, teknik yönden bakıldığında, daha kolay ve sık sürümler sayesinde daha fazla dağıtımcı ve daha küçük sürümler sunmak mümkün hale getirildi. Bu durum, kod dağılımını ve son kullanıcılar için daha fazla özellik aktarmayı kolay hale getirmiş oldu. Buna bağlı olarak takım daha küçük ve daha sık kod salınımının daha az riskli olduğunu ve daha az servis kesintisine neden olduğunu gözlemledi. Sürümlerin daha kolay ve sık bir şekilde salınmasını, yazılım geliştiriciler ve operasyon ekipleri arasında olan bilgi alışverişi sağladı.

DevOps'u benimsemeden önce, operasyon personeli, temelde yanlış bir şey olmadıkça, ürün ekiplerine geri bildirimde bulunmadan, sunuculara ve altyapıya bakarak sistem kontrolünü sağlayan yöneticilerdi. Bu geleneksel yaklaşımdan bulut barındırma platformlarına geçerek, operasyon ekiplerine otomasyon ve yapılandırma yönetimi yapabilme gücü kanıtlandı. Aynı zamanda da operasyon çalışanları kodun neden belirli bir şekilde yazıldığını anlamaya başladı ve bu da kaynaklanan hataların sebeplerini daha verimli altyapı çözümleri ile tasarlamalarına yardımcı oldu. Gelişim ve operasyonlar hakkında ortak bilgi sahibi olmanın yanı sıra ortak konumdaki bilgilere sahip olmak, geliştiriciler ve operasyonlar arasındaki iletişimin daha doğal ve daha zengin olduğu anlamına da geliyordu. Bunun en güzel örneği, şirket içinde artan yüz yüze iletişim ve düşen malileşmeler ile herkesin bir takım halinde hissetmesiydi. Sonuç olarak bir yıl gibi kısa bir sürede iki ekibin ortak iş birliği ile birçok problemli görülen alanlarda çözüm üretimi mümkün hale getirildi.

BÖLÜM 6. TARTIŞMA VE SONUÇ

Bu araştırmada, eski olarak tanımlanabilecek uygulama geliştirme disiplinleri ve modellerine sahip kuruluşların bu sistemlerden yeni oluşturulmuş sistemlere geçişleri hakkında detaylı bir inceleme yapılmıştır. İnceleme süresince izlenen süreçlerde öncelikle tanımlamalardan başlanılmıştır. Gerek duyulan tanımlamaların ardından yeni proje geliştirme modellerinin eski modellere göre karşılaştırması yapılmıştır. Sonrasında yaşanan en yaygın sorun olan eski sistemlerden yeni sistemlere geçişin zorlukları ve bu zorlukların nasıl çözüldüğü bulunan tecrübeler kaynakça gösterilerek anlatılmıştır. Sonuç olarak araştırma, DevOps geçişlerinde eski yapılı kurum kültürlerinin karşılaştığı zorlukların nasıl aşılabileceğini cevaplamıştır. Aynı zamanda uygulamalar ile birlikte geçişlerdeki verim artışlarının hangi başlıklar altında incelenmesi gerektiği vurgulanmıştır.

Bu çalışmanın sonucuna göre, eski yapılı proje yönetim modellerinden yeni modellere geçişin temel sebeplerinden birisi, modelin getirdiği esnek disiplin anlayışıdır. Özellikle günümüz marketinde gelişmelerin hızı ile birlikte gerek duyulan sürekli yenilenme ihtiyacına eski modellerin cevap verememesi üzerinde durulmuş ve bu nedenle yeni modellerin ihtiyacı açıklanmıştır. Ayrıca yeni modelin getirdiği kolaylık ve yararlar, belirli başlıklar altında anlatılmıştır.

Çalışma boyunca yeni modellerin uygulanabileceği iki alan verilmiştir. İlk olarak bahsedilen alan, yeni bir uygulama yapım aşamasıdır. Bu alan için yeni modellerin uygulanışı adımlar halinde anlatılmıştır. Bir diğer uygulama alanı ve tezin asıl konusunun kapsamı eskiden ve günümüze kadar çeşitli yazılım uygulamaları kullanarak gerçekleştirilmiş iş akışına sahip kuruluşların canlıdaki sistemlerinde gerçekleşmesi planlanan değişimdir. Bu durum aslında zor gözükmek ile birlikte belirli tecrübelere sahip büyük kuruluşların izlediği adımlar ve dikkat edilmesi

gereken temel hususlar ile birlikte kolaylaştırılmaktadır. Bu durum, çalışma boyunca çeşitli başlıklar altında değinilmiştir.

Çalışmanın son kısmında bahsedilen geçiş evrelerini daha önce tamamlamış küresel büyüklüklere sahip kuruluşların bu evreleri nasıl tamamladıklarına dair örnekler yer almıştır. Bu örnekler ile öncesinde yapılan karşılaştırma ve tanımlamalar uygulamalı olarak örneklendirilmiş ve anlatılmıştır.

Sonuç olarak çalışmadan elde edilen önemli bulgular şu şekilde özetlenebilir;

a. Eski Proje yaşam döngüleri ve proje geliştirme modelleri ile yeni modellerin karşılaştırılması ve modeller arasındaki pozitif ve negatif yanların oluşturulması.
b. Yeni modellere geçmenin önemi, sağladığı avantajlar ve bunların sonucunda kuruluşların yeni modeli tercih etme sebepleri
c. Yeni modele geçerken uygulanması gereken adımlar
d. Yeni modele geçerken kurumların yaşayabileceği sorunlar ve bu sorunların çözümlenmesi
e. Yeni modeli uygulamış küresel büyüklüğe sahip kuruluşların tecrübeleri

KAYNAKLAR

Amber, S. (2003). Agile database techniques: Effective strategies for the agile software developer. 2nd edn. Singapore: Wiley Publishing.

Amlani, R. (2012). 'Advantages and limitations of different SDLC models', International Journal of Computer Applications & Information Technology, vol (1/3). Erişim adresi: http://www.ijcait.com/IJCAIT/13/1334.pdf (Erişim tarihi: 18.03.2019)

Aytekin, A. Macit, Y. (2018). GelİşletDevOps) Yaklaşımında Konteyner Dönüşümü Deneyimi. Havelsan 2018 Akademik Yayınları. p140-145, Erişim adresi: www.havelsan.com.tr/akademik-yayinlar.pdf (Erişim tarihi: 16.07.2019)

Baker, C. (2015). What is DevOps and why does it matter? Erişim adresi: www.zdnet.com/article/what-is-devops-and-why-does-it-matter/ (Erişim tarihi: 18.03.2019)

Baukes, M. (2016). Configuration & Security Management for DevOps. Erişim adresi: www.upguard.com/blog/testing-avoid-configuration-security-issues (Erişim tarihi: 23.03.2019)

Beğendi, O (2018). DevOps Yaptık da n'oldu? Erişim adresi: www.medium.com/hesapkurdu-development/devops-yaptik-da-noldu (Erişim tarihi: 12.08.2019)

Bentley, M., Devita, J., & Will, V. (2015). Making the Transition to DevOps. Erişim adresi: www.docker.com/sites/default/files/DevOpspdf (Erişim tarihi: 15.03.2019)

Bradley, T. (2015). Moving from legacy to DevOps is a journey that takes time…and the right tools. Erişim adresi: www.devops.com/moving-from-legacy-to-devops-is-a-journey-that-takes-timeand-the-right-tools/ (Erişim tarihi: 05.04.2019)

Cois, C. A. (2014). A Generalized Model for Automated DevOps. Erişim adresi: www.insights.sei.cmu.edu/sei_blog/2014/06/a-generalized-model-for-automated-devops.html (Erişim tarihi: 08.04.2019)

Curtis, B., Lesokhin, L., Szynkarski, A., & Duthoit, S. (2014). The CRASH Report 2014-2015 Erişim adresi: www.castsoftware.com/resources/research-library (Erişim tarihi: 05.04.2019)

Dale, N. (2016). The Waterfall Model of Software Development. Erişim adresi: www.zeusdesign.com (Erişim tarihi: 23.03.2019)

Edler, M., Crume, J., Hahn, T., Pogue, S., & Sharma, S. (2014). Security considerations for DevOps adoption. Erişim adresi: www.ibm.com/developerworks/library/d-security-considerations-devops-adoption (Erişim tarihi: 16.03.2019)

Font, V. (2016). The Father of Waterfall Isn't. Erişim adresi: http://ultimatesdlc.com/father-waterfall/ (Erişim tarihi: 28.03.2019)

Fowler, M. (2018). Refactoring: Improving the design of existing code, 2nd edn. Addison-Wesley Professional.

Gajda, K. (2018). Build a DevOps culture and squads. Erişim adresi: www.ibm.com/garage (Erişim tarihi: 17.07.2019)

Garinchaud, N. (2012). What Exactly is DevOps. Erişim adresi: www.drdobbs.com (Erişim tarihi: 25.03.2019)

Gruver, G. and Mouser, T. (2015). Leading the Transformation Applying Agile and DevOps Principles at Scale, 1st edn. Portland: IT Revolution.

Gruver, G., Young, M., & Fulghum, P. (2012). A Practical Approach to Large-Scale Agile Development. 1st edn. Addison-Wesley Professional

Gülaçtı, E. (2017). Logo'nun DevOps Yolculuğu. Erişim adresi: blog.logo.com.tr (Erişim tarihi: 27.04.2019)

Hand, J. (2013). The IT Culture War: The Struggle to Adopt DevOps. Erişim adresi: www.wired.com (Erişim tarihi: 18.03.2019)

Hughey, D. (2009). The Traditional Waterfall Approach. Erişim adresi: www.umsl.edu (Erişim tarihi: 16.03.2019)

Hulm, G. (2015). Does DevOps hurt or help security. Erişim adresi: www.csoonline.com (Erişim tarihi: 05.04.2019)

Kerravala, Z. (2015). The shift to DevOps requires a new approach to security. Erişim adresi: www.networkworld.com (Erişim tarihi: 16.03.2019)

Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S., & Hage, J. (2014). How Do Professionals Perceive Legacy Systems and Software Modernization? Erişim adresi: www.servicifi.files.wordpress.com (Erişim tarihi: 24.03.2019)

Lauder, A. & Kent, S. (2000) Legacy System Anti-Patterns and a Pattern-Oriented Migration Response. Canterbury: Kent Academic Repository

Lauder, A. & Lind, M. (2007). Legacy Systems: Assets or Liabilities, Working paper in Boras Studies of Information Systems, No. 20. Sweden

Lester, R. (2013). The Three Most Important Agile Practices. Erişim adresi: www.scrumaliance.com (Erişim tarihi: 16.05.2019)

Levy, T. (2015). 9 Open Source DevOps Tools We Love. Erişim adresi: www.devops.com (Erişim tarihi: 22.03.2019)

Liu, Y., Li, C. & Liu, W. (2014). Integrated Solution for Timely Delivery of Customer Change Requests: A Case Study of Using DevOps Approach. International Journal of u- and e-Service, Science and Technology, 7 (2), p41-50

Logan, M. (2014). DevOps Culture Hacks. Erişim adresi: www.devops.com (Erişim tarihi: 23.03.2019)

Massey, V. Satao, K. (2012). 'Comparison various SDLC Models and the new proposed model on the basis of available methodology', International Journal of Advanced research in computer science and software engineering, vol (2/4). Erişim adresi: www.pdfs.semanticscholar.org (Erişim tarihi: 12.03.2019)

Mathaisel, B. (2013). A CIO's approach to resolving the agility-stability paradox. Erişim adresi: www.usblogs.pwc.com (Erişim tarihi: 18.03.2019)

Mircea, M. Stoica, M. Ghilic-Micu, B. (2013). 'Software Development: Agile vs. Traditional', Informatica Economica, vol (17/4), p64-76

Mohamed, A. (2016). Open source software security. Erişim adresi: www.computerweekly.com (Erişim tarihi: 20.03.2019)

Mohamed, S. I. (2015). 'DevOps shifting software engineering strategy value-based perspective', IOSR Journal of Computer Engineering, (Vol 17, Issue 2). Erişim adresi: www.scholar.msa.edu.eg (Erişim tarihi: 16.03.2019)

Murphy, P. (2009). How Do You Define a "Legacy" Application? Erişim adresi: www.go.forrester.com (Erişim tarihi: 22.03.2019)

Nemmers, J. (2015). How to do DevOps Without Leaving Legacy Behind. Erişim adresi: www.ansible.com (Erişim tarihi: 17.03.2019)

Novak, A. (2014). 6 Elements of Highly Successful DevOps Environments. Erişim adresi: www.blog.newrelic.com (Erişim tarihi: 08.04.2019)

Null, C. (2015). The 4 biggest reasons for DevOps failure. Erişim adresi: www.techbeacon.com (Erişim tarihi: 05.04.2019)

Paul, M. (2014). The Measurable and Important Benefits of DevOps. Erişim adresi: www.logicworks.com (Erişim tarihi: 02.04.2019)

Radack, S. (2009). The System Development Life Cycle (SDLC). Erişim adresi: www.csrc.nist.gov (Erişim tarihi: 05.04.2019)

Ravichandra, A. (2015). Do Regulatory Auditors Hate Continuous Delivery? Erişim adresi: www.devops.com (Erişim tarihi: 23.03.2019)

Reminnyi, S. (2015). Developing Test Automation Scripts and Automation Frameworks. Erişim adresi: www.infoq.com (Erişim tarihi: 28.03.2019)

Riley, C. (2014). 6 Challenges Facing DevOps and Operations Teams in 2015. Log entries. Erişim adresi: www.dzone.com (Erişim tarihi: 26.03.2019)

Robinson, A. (2015). 'Continuous Security: Implementing the Critical Controls in a DevOps Environment', SANS Institute Information Security Reading Room. Erişim adresi: www.sans.org (Erişim tarihi: 27.04.2019)

Rubio, D. (2015). New Jenkins Plugin Identifies Known Security Vulnerabilities in Open Source Projects. Erişim adresi: www.blog.openhub.net (Erişim tarihi: 12.06.2019)

Saran, C. (2015). IT faces uphill struggle in DevOps culture change. Erişim adresi: www.computerweekly.com (Erişim tarihi: 05.06.2019)

Soylu, S. (2017). 'Kamu kurumlarında yazılım yaşam döngülerinin uygulanabilirliği ve Çevre ve Şehircilik Bakanlığı için öneriler', Çevre ve şehircilik bakanlığı uzmanlık tezi. Erişim adresi: www.webdosya.csb.gov.tr (Erişim tarihi: 08.07.2019)

Şenyurt, B. (2018). DevOps Eğitiminden Aklımda Kalanlar. Erişim adresi: www.medium.com (Erişim tarihi: 28.04.2019)

Şuman, N. (2018). İş Bankası İlkleri Yaşatmada Yeni Hikayeler Yazacak. Erişim adresi: www.platinonline.com (Erişim tarihi: 05.07.2019)

Trimble, J. (2013). User Centered Agile Development at NASA – One Groups Path to Better Software. Erişim adresi: www.slideshare.net (Erişim tarihi: 22.07.2019)

Trimble, J. Shirley M. Hobart, S. (2016). Agile: From software to Mission system. Erişim adresi: www.ntrs.nasa.gov (Erişim tarihi: 14.08.2019)

Waits, T. (2015). Three Challenges to Documentation for DevOps Teams. Erişim adresi: www.insights.sei.cmu.edu (Erişim tarihi: 21.03.2019)

Wallgren, A. (2016). Two key challenges of using open source in the enterprise. Erişim adresi: www.betanews.com (Erişim tarihi: 23.06.2019)

Webster, C. (2012). Delivering software into NASA's Mission Control Center using agile development techniques. Erişim adresi: www.ieeexplore.ieee.org (Erişim tarihi: 26.07.2019)

Wurst, N. (2015). Testing Will Never Be Automated. DevOps. Erişim adresi: www.devops.com (Erişim tarihi: 05.08.2019)

ÖZGEÇMİŞ

Ömer Furkan Arifoğlu, 01.02.1993'de Sakarya'da doğru. İlk, orta ve lise eğitimini Sakarya'da tamamladı. 2011 Yılında Figen Sakallıoğlu Anadolu Lisesinden mezun oldu. 2011 Yılında başladığı Sakarya Üniversitesi Bilgisayar Mühendisliği Bölümü'nü 2015 yılında bitirdi. Bölümü bitirene kadar Sakarya Üniversitesi de dahil olmak üzere çeşitli firmalarda yazılım geliştiriciliği rolü üstlendi. 2017 Yılında başladığı Bilgisayar ve Bilişim Mühendisliği Yüksek Lisans programında öğrenimine devam etmektedir. Lisansı bitirdikten sonra sırasıyla Borusan Lojistik Firmasında Arge Departmanında Yazılım Uzmanı, Türkiye Finans Katılım Bankasında Arge ve Framework Departmanında Yazılım Kıdemli Uzmanı olarak görev yaptı. Halihazırda da HCL Technologies firmasında Teknik Lider olarak görev yapmaktadır.