TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Hannes Karask 142112IABM

PIDEV INTEGREERIMINE JA TARNE WINDOWSI JA .NET PÕHISES HAJUSSÜSTEEMIS

Magistritöö

Juhendaja: Jaak Tepandi
Professor

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Hannes Karask

08.05.2017

Annotatsioon

Agiilsete arendusmetoodikate peavoolu jõudmisega on rakenduse tarnetsükkel läinud oluliselt lühemaks. Sageda tarnimisega kaasnevad suuremad riskid ning nende juhtimiseks on vaja efektiivset ning veakindlat protsessi.

Käesoleva magistritöö eesmärgiks on välja töötada tõhus pideva integreerimise ja tarne protsess tarkvarauuenduste jaoks. Eesmärgist lähtuvalt analüüsitakse tarkvara arendusprotsessi pideva integreerimise ja paigalduse vaatenurgast ning luuakse näidissüsteemi põhjal efektiivsem protsess.

Magistritöö teoreetilises osas selgitatakse probleemi tausta ning olemasolevaid tehnilisi lahendusi ja nende kitsaskohti. Praktilises osas võetakse protsessi disainimise aluseks kompleksne Windowsi platvormile .NET-põhiste vahenditega loodud hajussüsteem. Töö autor analüüsib näidissüsteemi ning püstitab hüpoteesid parendusettepanekutega. Tulemusteni jõudmiseks analüüsib autor protsessi erinevaid samme ning leiab empiirilisel teel optimaalseima viisi protsessi täiustamiseks.

Lõputöö tulemusteks on efektiivse pideva integreerimise ja paigaldamise protsess, realisatsioon ning selle käigus tekkinud juhised. Lisaks selgitab autor ka eelpool mainitud protsessi eri etappe ning analüüsib läbiviimiseks erinevaid alternatiive.

Käesoleva magistritöö tulemusena valminud protsessijuhistest on kasu kõigil, kes puutuvad kokku tarkvaraarendusega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 61 leheküljel, 13 peatükki, 16 joonist, 2 tabelit.

Abstract

Continuous Integration and Delivery Based on .NET Distributed Systems in Windows

Agile software methodologies have become mainstream and caused significant reduction in software development cycle. Companies want to deploy software faster in order to adapt to the rapidly changing environment and requirements. Faster deployments mean less time for testing the product and consequently heightened risk.

The aim of this master's thesis is to develop an efficient process for continuous integration and delivery for software projects. It is important to analyse the software development process as a whole from the continuous integration and deployment point of view. The key characteristics of the new process will be speed, predictability and reliability. Software development must be a quick and repeatable undertaking. The new process will be developed based on an existing process, which handles the deployment of a Windows .NET distributed system for a SaaS (Software as a Service) solution.

The author begins by providing background information about related topics, continuing to conduct an analysis on the existing process and provide hypothetical improvement suggestions. The author will then inspect different steps involved in the process and find the optimal solution through empirical analysis.

This thesis results in three separate artefacts: a new efficient continuous integration and delivery process, an implementation of this process and instructions for the future and existing processes. The study may benefit anyone who is involved in the development of software, including companies across different sectors and sizes.

The thesis is in Estonian and contains 61 pages of text, 13 chapters, 16 figures, 2 tables.

Lühendite ja mõistete sõnastik

.NET	*Dot Net*, Microsofti poolt loodud tarkvararaamistik
.sln	Visual Studio struktuurifail
AB	Andmebaas
Agiilne arendus	*Agile Development*, tarkvaraarendus metoodika
API	*Application programming interface*, rakendusliides moodulina või eraldiseisva veebisaidina mis võimaldab erinevaid rakendusi endaga liidestada
ASP.NET MVC	*Active Server Pages Model View Controller*, Microsofti veebiraamistik mis järgib tuntud tarkvaramustrit Mudel Vaade Kontroller
C#	Microsofti loodud kõrgtaseme programmeerimiskeel
DDL	*Data definition language*, andmete struktuuri kirjeldamise keel
DevOps	*Development + Operations*, praktika milles arendus- ja ülalhoiumeeskonnad töötavad koos, et efektiivsemalt tarnida tarkvara
DLL	*Dynamic-link library*, Microsofti tehnoloogiatel põhinev programmimoodul
Docker	Tarkvaraline konteinerdamise platvorm
GET päring	HTTP päringu tüüp
Git	Hajus versioonihaldustarkvara
ESB	*Enterprise service bus*, teenusesiin
Veebifond	*Application Pool*, IIS veebiserveril käitava rakenduse konteiner
HTTP	*HyperText Transfer Protocol*, hüperteksti edastuse protokoll
IDE	*Integrated development environment*, rakenduste loomiseks loodud tarkavara
IIS	*Internet Information Services*, Microsofti poolt loodud veebiserver
JSON	*JavaScript object notation*, JavaScripti objektide notatsioon
Kiirpaik	*Hotfix*, kiirkorras loodav paik vea kõrvaldamiseks
Koormusjaotur	*Load balancer*, pöördproksina töötav seade, mis jaotab võrgu- või rakenduseliiklust mitmele serverile, suurendades teenuse jõudlust ja käideldavust

LDAP	*Lightweight Directory Access Protocol*, kataloogipöörduse kergprotokoll
MS	*Microsoft Corporation*, tarkvaraettevõte
MSBuild	Microsofti rakendus .NET rakenduste ehitamiseks
MSDeploy	Sama mis Web Deploy, käsurea utiliidi nimi on MSDeploy.exe
MSpec	.NET raamistikul põhinev konteksti põhine ühiktestide raamistik
MySQL	Relatsiooniline andmebaas
NuGet	Microsofti arendusplatvormi keskne paketihaldur
NUnit	xUnit tüüpi .NET raamistikule loodud ühiktestiraamistik
PI	Pidev Integreerimine (ingl *Continuous Integration*)
PowerShell	Skriptimiskeel ja keskkond
SaaS	*Software as a Service*, tarkvara kui teenus
SDK	*Software Development Kit*, programmipakett, mis võimaldab programmeerijal luua rakendusi konkreetsele platvormile
SQL	*Structured Query Language*, struktureeritud päringute keel
zip	Tihendatud fail
7z	Tihendatud fail
TDD	*Test-driven development*, arenduspraktika milles enne funktsionaalsuse kirjutamist luuakse test
TeamCity	Pideva integratsiooni servertarkvara
Tehis	*Artifact*, vahe- või kõrvalsaadus tarkvaraarenduses
Toodang	*Production*, keskkond milles on lõppkasutajatele kättesaadav versioon rakendusest
VCS	*Version control systemi*, versioonihaldustarkvara
Visual Studio	Microsofti tehnoloogiate arendemiseks loodud IDE
Web Deploy	Microsofti rakendus veebirakenduste paigaldamiseks IIS rakendusserverisse
WinRM	*Windows Remote Management*, Microsofti WS-Management protokolli realisatsioon. Võimaldab PowerShellil ühenduda väliste serveritega
Ülalhoid	*Operations*, IT teenuste haldamisega seotud protsessid
XML	*Extensible Markup Language*, laiendatav märgistuskeel

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Tarkvaraarenduse protsess on läbinud pika arengu ning muutunud ajas küpsemaks ja keerulisemaks. Oluliselt on lühenenud ajaperiood, mille jooksul tarkvaramuudatus jõuab lõppkasutajani. On ettevõtteid, kes tegutsevad tundmatul maastikul ning nende ärimudel on üles ehitatud pidevale katsetamisele. Eriti näeb sellist tendentsi alustavate ettevõtete puhul, kes oma äriideed alles valideerivad. Samas võivad sarnaselt talitada ka vanemad ettevõtted, kes soovivad konkurentsis püsida. Tarkvarauuenduste lansseerimisel on alati riskid, mis võivad avalduda programmivigadena või isegi turvaaukudena ja võivad ettevõttele kalliks maksma minna.

Tarkvara arenduse ja tarne protsess on olemuselt keerukas ning see vajab kindlate protsesside ning distsipliini järgimist. Iga infosüsteem eeldab selle tarbeks kohaldatud keerulist protsessi, mille täitmisel on lihtne teha vigu. Selline protsess hõlmab enamasti erinevate serveritega suhtlust ning kindlas järjekorras tegevuste sooritamist. Ebaefektiivne protsess on otseses või kaudses mõttes ettevõttele kahjumlik ning ebaefektiivsusi on keeruline tuvastada. Käesolev töö analüüsib efektiivset tarkvara arenduse protsessi muudatuse integreerimise ja paigaldamise vaatenurgast Windowsi platvormil ning kirjeldab, kuidas erinevaid etappe optimeerida.

1.1 Taust ja probleem

Mõnda aega tagasi oli loomulik tarkvara uuendusi teha regulaarselt kord aastas või isegi kahe jooksul, kuid nüüd on tänu agiilsete (*agile development*) metoodikate peavoolu jõudmisega arendustsükli kestus kordades vähenenud. Tuntud veebibrauser Chrome tarnib põhi (*major*) versioone kaheksa korda aastas (vt peatükk 2) ning sotsiaalmeedia gigant Facebook on 2012. aastal väitnud, et tarnib koguni kaks korda päevas [1].

Kiire ja tõhus tarnimine eeldab efektiivset protsessi, mis peab viima uue tarkvara lansseerimisega seotud riskid minimaalseks. Kui ettevõte tarnib oma tarkvara harva, siis manuaalne paigaldamise protsess ei pruugi olla automaatsega võrreldes kuigi riskantne.

Kuid kui tarnevälde on lühike, on manuaalse protsessi korral riskid juba märgatavalt suuremad.

Enamik töös kirjeldatud praktikaid on rakendatavad nii Linuxi kui ka Windowsi keskkondades. Siiski, leidub teatavaid erisusi, mis on käesoleva töö kirjutamise hetkel ühe või teise keskkonna põhised. W3Techs väitel on Unixi ja Windowsi turuosad vastavalt 66,6% ja 33,4% [2]. On teada, et paljud ettevõtted peidavad veebiserveri poolt väljaantavaid HTTP päiseid, mistõttu ole päris täpselt teada, kui suur vastav jaotus tegelikult on.

Uuringust on näha, et Windows platvormil on märkimisväärne turuosa. Sellest tulenevalt keskendub autor oma töös sellele platvormile, et analüüsida, milline võiks olla optimaalne tarkvara tarnimise töövoog ning pakkuda välja efektiivseks tarneprotsessiks sobivad praktikad. Pidev integreerimine ja paigaldus on infosüsteemi ulatuse kasvades üha olulisem osa arendusprotsessist. Mida suuremaks kasvab infosüsteem, seda keerukamaks muutub ka pideva integreerimise ja paigalduse protsess. Töös uuritakse, milliseid valikuid protsessi ülesseadmisel teha, et muuta keeruline tarneprotsess efektiivseks ning juhtida sellega kaasnevaid riske.

Magistritöö tulemusena valminud protsessidest on kasu kõigile, kes tegelevad tarkvara arendamise ja tarnimisega, sõltumata projekti suurusest ja ettevõtte tegevusalast. Suurimat kasu võiksid antud töö käigus disainitud metoodikatest saada aga SaaS (Software as a Service) põhise tootega ettevõtted, kuna paljud töös kirjeldatud metoodikad põhinevad näidissüsteemist tulenevalt serveripoolsel tarkvaral.

1.2 Ülesande püstitus

Käesoleva magistritöö eesmärgiks on keerukama näidissüsteemi analüüsimisel selgitada välja levinumad kitsaskohad pideva integreerimise ja paigaldamise protsessis. Seejärel otsib autor leitud probleemidele lahendused ning disainib nende põhjal uue efektiivsema protsessi, mis võiks olla aluseks tulevastele ja olemasolevate Windows platvormi .NET põhistele SaaS tarkvara arendusprotsessidele.

1.3 Metoodika

Esimeses osas teostab autor näidissüsteemi analüüsi ekspertteadmiste põhjal, tuues välja hüpoteesid, millele järgmistes osades üritatakse kinnitust leida. Seejärel kirjeldab autor protsessi iga sammu juures teoreetilist poolt, levinud parimate praktikate näitel ning leiab empiirilisel teel kvantitatiivset analüüsi kasutades parima lahenduse.

1.4 Ülevaade tööst

Teises peatükis annab autor lühike ülevaade agiilsete arendusmetoodikate laialdase kasutuselevõtu ning rakenduse tarnetsükli lühenemise seostest.

Kolmas peatükk seletab lahti lendsõna DevOps, selle seose arenduse ning ülalhoiuga ja selgitab selle rolli arendusprotsessis. Räägitakse ka erinevatest serverite seadistamise võimalustest ning Dockeri rollist Windowsi maailmas.

Neljas peatükk räägib monoliitide ja mikroteenuste rolli tähtsusest rakenduse erinevates elutsüklites ning selgitab, millal võiks eelistada ühte või teist.

Viiendas peatükis lõpetatakse sissejuhatavate taustteemadega ning alustatakse näidissüsteemi analüüsi. Vaatluse alla võetakse üks Windows platvormil olev .NET raamistikul loodud SaaS tüüpi rakendus ning selle integratsiooni- ja tarneprotsess. Analüüsitakse selle kitsaskohti ning luuakse hüpoteesid parendustele.

Kuuendas osas selgitatakse lugejale pideva integreerimise olemus, tutvustatakse pideva integreerimise servertarkvara TeamCityt ning modelleeritakse näidissüsteemi põhjal uus ja parem protsess. Viimaks vaadatakse viise, kuidas võiks pideva integreerimise protsessi siduda projektijuhtimise- ning suhtlustarkvaraga.

Seitsmendas osas keskendutakse lähtekoodi ehitamise etapile ning selle optimeerimisele. Kirjeldatakse lühidalt ehitusserveri ülespanekut ning ehitamist pideva integreerimise vaatenurgast. Käsitletakse MSBuildi eripärasid ning mõju ehitamiskiirusele. Leitakse ja kirjeldatakse, kui palju erinevad MSBuildi parameetrid mõjutavad ehitamistulemusi. Seejärel räägib autor lühidalt versiooninumbri genereerimisest ning veebisaidi ehitamise spetsiifikast.

Kaheksandas osas räägitakse konfiguratsioonihaldusest pideva integreerimise ning paigaldamise protsessis ning leitakse optimaalseim viis tehiste transportimiseks.

Üheksandas osas vaadatakse automaattestimise protsessi näidissüsteemis ning optimeeritakse ühik- ning vastuvõtutestide käitamise kiirust.

Kümnendas osas selgitatakse veebisaidi efektiivset paigaldamise protsessi rakendusserverisse IIS ning selle automatiseerimist skriptimisskeele PowerShell abil. Seejärel räägitakse analoogselt Windows teenuste paigaldamisest ning andmebaasi migratsioonidest.

Üheteistkümnendas osas võrreldakse pidevat tarnet ja pidevat paigaldamist ning leitakse millises kategoorias näidissüsteem asub.

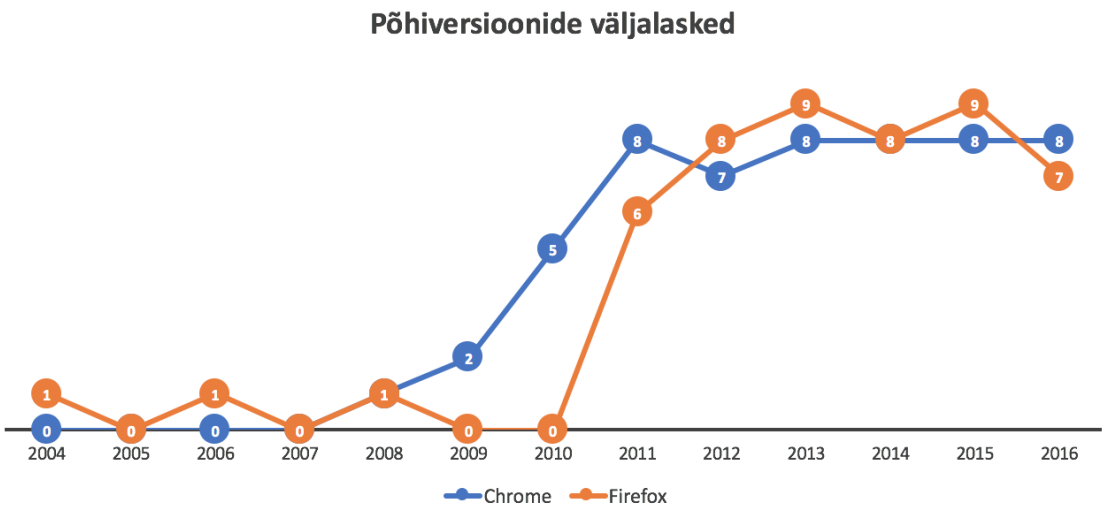
Kaheteistkümnes peatükk võtab kokku töö tulemused, võrreldakse saavutatud tulemusi analüüsi osas seatud hüpoteesidega. Lõpuks pakutakse välja võimalikud sammud protsessi edasiseks optimeerimiseks.

2 Agiilne arendus

Seoses agiilse arenduse (*agile development*) populaarseks muutumisega on tarkvara uuenduste lansseerimine muutunud väga sagedaks. Kui kosemeetodi puhul võis ühe tarkvara versiooni loomiseks kuluda kuid või isegi aastaid, siis agiilse arenduse puhul lansseeritakse uuendus reeglina iga nädala või kahe tagant (sõltudes arendustsükli ehk sprindi pikkusest).

Agiilse arenduse manifesti põhimõtetes on selgelt välja toodud: „Tarnime tarkvara nii tihti kui võimalik, soovitavalt iga paari nädala kuni paari kuu tagant.“, mis toetab põhimõtet: „Kõige olulisem on tagada kliendi rahulolu, tarnides talle vajalikku tarkvara võimalikult kiiresti ja tihti“ [3].

Selgelt on näha lühenenud arendustsükleid veebibrauserite maailmas, kus pikka aega olid versiooninumbrid alla viie ning siis lühikese ajaga juba 20 ja üle (vaata Joonis 1) [4] [5].



Joonis 1. Chrome ja Firefox põhiversioonid.

Jooniselt on näha, et enne 2011. aastat lasi FireFox keskmiselt välja ühe põhi (*major*) versiooni kahe aasta jooksul. Pärast 2011. aastat on ühes aastas keskmine põhiversioonide arv kaheksa ringis. Chrome tuli küll tootega veidi hiljem turule, aga graafikult on näha selgelt sarnast trendi FireFoxi lansseerimise strateegiaga.

Agiilset arendust nimetatakse ka tihti paindlikuks (*lean*) arenduseks. Nimetus „paindlik“ on pärit tootmismaailmast ning selle võttis 1950. aastatel kasutusele autotootja Toyota,

kes suutis oma tootmisprotsessi teha märkimisväärselt efektiivseks. Paindlikkus tähendab üldisemalt kõige ebavajaliku eemaldamist ning kiiret reageerimist välisele keskkonnale [6].

Agiilse arenduse metoodikad on sobilikud nii väiksematele kui ka suurematele ettevõttele, hoides kokku aega ning teisi ressursse. Väiksemate ettevõtete puhul võib algul tunduda, et eelpool nimetatud praktikate kasutusele võtmine võib hoopis tekitada protsessides liiasust, kuid pikas perspektiivis tasub see end ära [7, lk 26].

Agiilse arenduse eelduseks on teatava distsipliini järgimine, mida lihtsustab märgatavalt heade tööriistade kasutamine. Häid vahendeid agiilse arendusprotsessi toetamiseks, nii tasuta kui tasulisi, on saadaval üsna palju. Üks tuntumaid selles vallas on näiteks Jira vahendite komplekt.

3 DevOps

Tänapäeval levib enamus tarkvara üle interneti, mistõttu on pidevad tarkvarauuendused võimalikud ning enamasti ka ootuspärased, olenemata kas tegu on veebi-, mobiili- või töölauarakendusega. Paljud levinud tarkvaralised teenused on abonemendi (*subscription*) põhised ning järjest enam omavad pidevad toote iteratsioonid suuremat strateegilist tähtsust. Ajalooliselt on arenduse ja ülalhoiu (*operations*) vahel olnud üsna selge piir, mis on tänu agiilsele arendusele ja headele tööriistadele hajumas. Tekkinud on lendsõna DevOps [8].

Arendus on oma olemuselt pidevalt eksperimenteeriv, proovides järk-järgult tuua toodetesse sisse keerukust. Ülalhoiu olemus on arendusega vastuoluline, eeldades süsteemi pidevat toimimist ehk stabiilsust [8].

Kui arendusmeeskond jagab vastutust infosüsteemi hooldamisel terve elutsükli vältel, jagavad nad ka ülalhoiu meeskonna valupunkte, tuvastades kohti mida saaks arendusega parendada (näiteks efektiivsem logimine). Üleandmise periood ja mahukas dokumentatsioon ei toimi nii efektiivselt kui meeskondade algusest peale külg-külje kõrval koos töötamine. DevOpsi kultuur hägustab piiri arendajate ja ülalhoiu vahel ning võib tulevikus piirid isegi kaotada [9].

Viimasel ajal on palju kõlapinda saanud tarkvara Docker, mis muudab infosüsteemi infrastruktuuri haldamise oluliselt lihtsamaks. Suuremahuliste virtuaalmasinate asemel on nüüd väikesemahulised konteinerid, mille loomine ja käivitamine on kiire. Kahjuks on seda tüüpi konteinerlahendused hetkel veel suuresti vaid Linuxi platvormi põhised. Microsoft alles arendab konteinerdatud operatsioonisüsteemi ning julgematel on võimalus testida Windows Server 2016 Core või Nano versiooni. Microsoft on võtnud ka suuna arendada .NET raamistik multiplatvormseks, et kasutajatel oleks võimalus käitada oma rakendust vabalt valitud operatsioonisüsteemil. Sarnaselt konteineritega on ka see veel üsna algeline ning pidevas muutumises, mis muudab selle kasutamise tootmiskeskkonnas riskantseks.

Uue (testimis)keskkonna seadistamine võiks olla automatiseeritud, et vajaduse korral saaks kiirelt võimalikult sarnaseid keskkondi juurde luua. Ideaalsel juhul peaks saama uue testkeskkonna luua ühe nupuvajutusega. Käsitsi masinate haldamine võib kaasa tuua

ridamisi probleeme: ühes keskkonnas tekkinud viga on raske reprodutseerida teises keskkonnas, klastris unustatakse mõni masin uuendada ning puudub ajalugu tehtust. Üks näide siinkohal oleks lokaatide (*locale*) erinevused, mis mõjutavad näiteks kuupäeva või murdarvu kuvamist ja võib kehvalt tehtud rakenduse töös põhjustada ootamatusi. Fowler toob oma artiklis välja, et ainuke viis tõkestada masina konfiguratsiooni „triivimist“ on keelata ära otse ligipääs masinatesse (konsool või siis kaugtöölaud) ning lubada muudatuste tegemine ainult kindlal tarkvaral [10]. Tuntuimad rakendused, mis suudavad etteantud juhiste alusel servereid seadistada, on Puppet, Chef ja neist noorim, Powershelli põhine DSC (*Desired State Configuration*). Nendele antavad instruktsioonid saab lihtsalt versioonida ning seetõttu on auditeerimisel võimalik saada hea ülevaade masinaga seonduvast ajaloost. Eelpool mainitud tarkvarad suudavad ka konfiguratsiooni jälgida ning muutuse avastamisel viia see tagasi soovitud seisu.

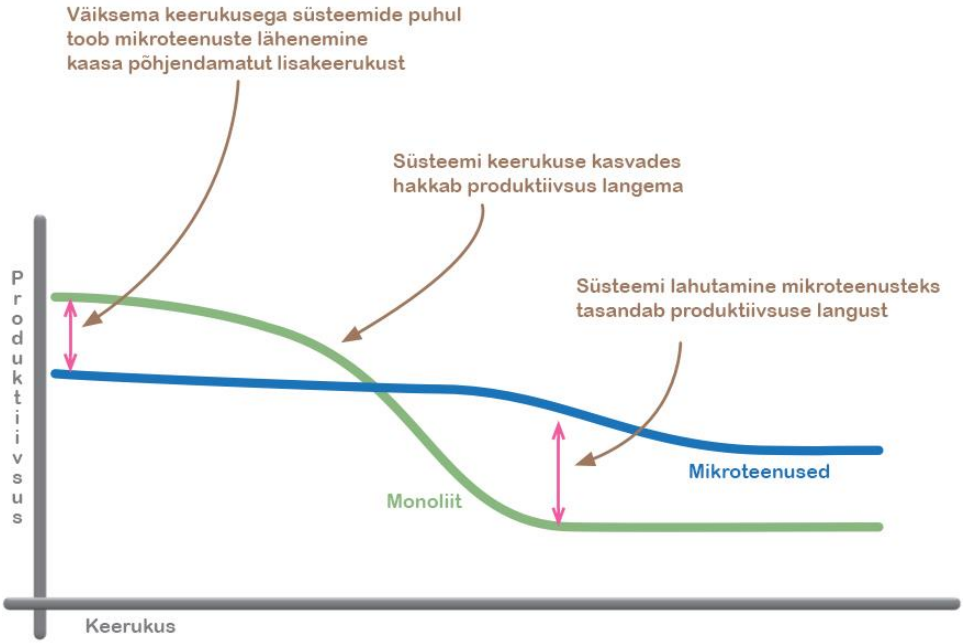
Hea näide elastsest arhitektuurist on Netflixi Chaos Monkey, mis üritab halvata osa infosüsteemist peatades teenuseid. Sellise protsessi kasutamine paljastab infosüsteemi kitsaskohad ning aitab üles seada veakindlama protsessi. Kui järgmine kord peakski juhtuma, et kell kolm öösel toimub rike, siis suure tõenäosusega suudab süsteem sellest ise taastuda nii, et tehnik ei pea sekkuma [11].

Üsna levinud on ka operatsioonisüsteemi tasandil loodud tõmmiste (*image*) kasutamine virtuaalmasinates. Stabiilsest serverist luuakse baastõmmis, mis omab vajalikku konfiguratsiooni, et sellest paljundada uusi servereid. Probleem tekib siis, kui ilmneb, et on vaja midagi muuta või uuendada. See eeldab kõikide samast tõmmisest loodud instantside üle käimist ning uue baastõmmise loomist. Kasutades eelpool mainitud masina konfiguratsiooni haldamise tarkvara, on protsess tunduvalt optimaalsem.

4 Monoliidid ja mikroteenused

Üldiselt algab uue infosüsteemi areng ühest suurest terviksüsteemist ehk monoliidist, millega valideeritakse süsteemi ja äriidee sobivus turu jaoks. Aja jooksul kasvab nii koodibaas kui ka arendusmeeskonna suurus. Suure monoliitse süsteemi arendamine muutub keerukamaks, kuna erinevate meeskondade muudatused ristuvad ja võivad tekitada konflikte. Suure monoliitse süsteemi testimine võib samuti olla murekoht, kuna arendajatel ei ole täpselt teada, millist osa koodimuudatus võib mõjutada ning mida täpselt testida tuleks. Täismahus testide komplekti käitamine võib võtta liiga kaua aega ning testide korrastamine on samuti aeganõudev. Sellest tulenevalt on iga järgneva muudatuse integreerimine süsteemi järjest tülikam. Suure monoliitse süsteemi puhul on keeruline ka vastutuse jagamine. Kui süsteemis esineb ebakõlasid, ei pruugi alati olla selge, kes probleemi lahendamisega tegelema peaks.

Mikroteenuste kasutusele võtmine toob kaasa lisakeerukust, mis ei ole väikese infosüsteemi puhul otstarbekas. Infosüsteemi arendav meeskond peab ühiselt ära tabama momendi, mil mikroteenuste kasutusele võtmine on õigustatud (vaata Joonis 2).



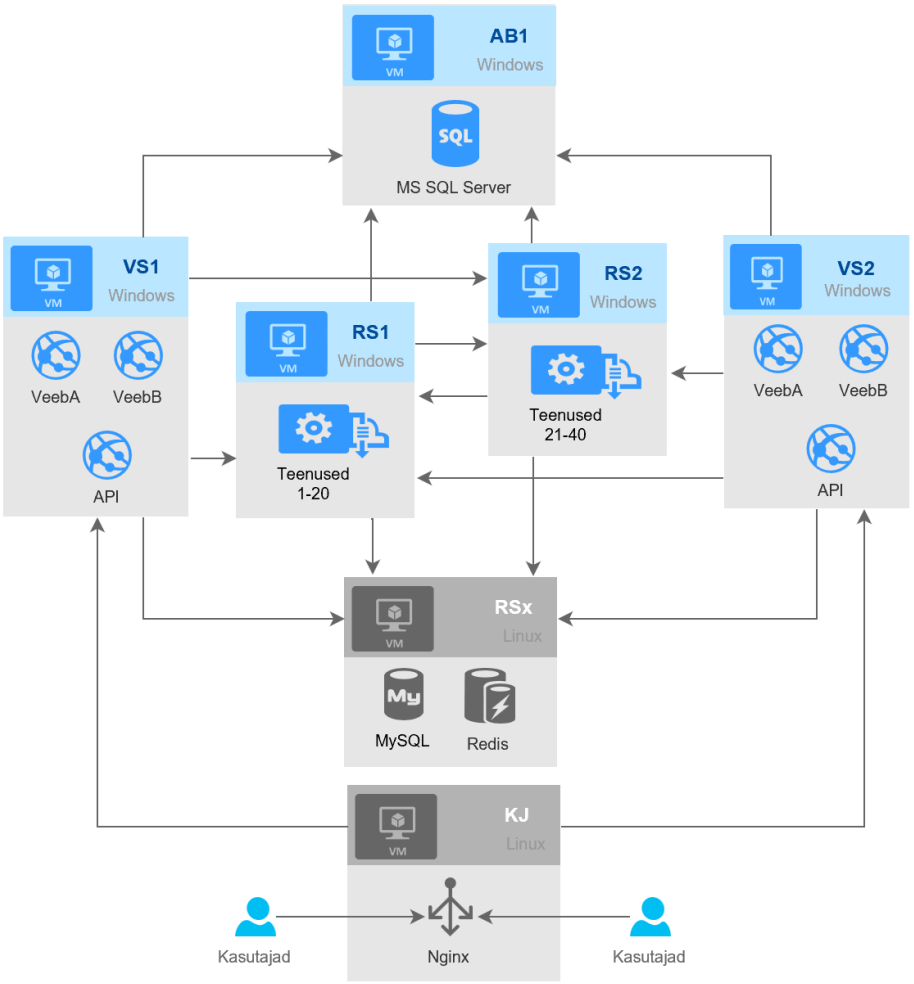
Joonis 2. Süsteemi keerukus monoliidi ja mikroteenuste põhises süsteemis [12].

Mikroteenused on väiksemad alamsüsteemid, mis suudavad üksteisega suhelda ning luua seeläbi ühise terviku. Lõppkasutaja ei saa sageli arugi, et tema rakendust teenindab tegelikult rohkem kui üks süsteem. Mikroteenused suhtlevad omavahel kasutades OSI mudeli järgi transpordi- või rakenduskihti, näiteks HTTP-d või otse läbi TCP [13].

Mikroteenuste kasutamisel tuleb leida mõistlik piir koodi- ja ülalhoiu keerukuse vahel. Mida rohkem teenuseid, seda keerulisem on nende paigaldus ja seire. Mikroteenuste kasutamise eelis on, et nad võimaldavad hajutada koormust erinevate füüsiliste või virtuaalsete serverite vahel. See omakorda võimaldab kaitsta ärikriitilisi protsesse ning tulla toime ajutise koormuse kasvuga. Teenuste kapseldamine eraldi osisteks annab võimaluse kasutada erinevaid tehnoloogiaid ja sellest tulenevalt ei ole ettevõte piiratud ainult üht tüüpi pädevusalaga (näiteks .NET). Mitmed ettevõtted kasutavad juba samas infosüsteemis eri tehnoloogiaid ning eri taustaga arendajaid, mis annab neile paremad väljavaated tööjõu leidmisel. Ühtlustamaks tehnoloogiatevahelisi piire, on loodud erinevaid lahendusi ühise liidese loomiseks, näiteks Apache Thrift.

5 Näidissüsteem

Käesolev peatükk kirjeldab ühe SaaS tüüpi rakenduse topoloogiat ning pideva integreerimise ja paigaldamise protsessi. Vaatluse all olev infosüsteem on ehitatud valdavalt Microsofti tehnoloogiatel ning koosneb ASP.NET MVC veebisaitidest (mida käitavad IIS rakendusserverid) ning taustal käivatest Windows teenustest. Teenused suhtlevad omavahel läbi teenusesiini (ESB, *Enterprise Service Bus*) NServiceBus. Järgnevalt on kujutatud infosüsteemi eri osade paiknemine serverites (vaata Joonis 3).



Joonis 3. Infosüsteemi topoloogia.

Nooled olemite vahel märgivad suhtluse suunda, KJ > VS1 tähendab, et koormusjaotur pöördub veebiserveri VS1 poole, aga mitte vastupidi. Jooniselt on lihtsuse mõttes ära jäetud analoogse ülesehitusega testkeskkond. Infosüsteem on umbes 7 aastat vana,

alustades kunagi monoliidina ning nüüd jaotatud mikroteenuste vahel laiali. Joonisel 3 on näha 5 Windows serverit (märgitud sinisega):

- Veebiserver VS1: IIS veebiserveris majutatavad ASP.NET veebisaidid VeebA, VeebB ja API.
- Veebiserver VS2: Dubleerib veebiserverit VS1, käideldavuse tõstmiseks.
- Andmebaasiserver AB1: Relatsiooniline andmebaas MS SQL Server.
- Rakendusserverid RS1 ja RS2: mõlemas 20 Windows Service teenust, mis kasutavad teenusesesiini NServiceBus.

Hallid segmendid joonisel on Linux serverid, mis omavad süsteemis toetavat funktsiooni:

- Rakendusserver RSx: MySQL relatsiooniline andmebaas failide hoidmiseks ja võti-väärtus andmebaas Redis puhverdamiseks.
- Koormusjaotur KJ: HTTP server NGINX, pöördproksi, mis jaotab koormust veebisaitide vahel. Kõik päringud, mis veebisaitidele tehakse, käivad koormusjaoturist läbi ning suunatakse allavoolu olevatele veebisaitidele masinas VS1 ja VS2.

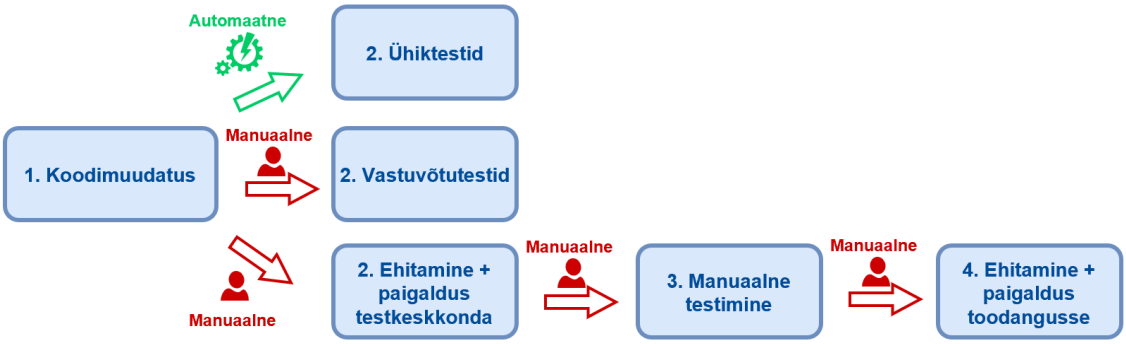
Linuxi masinatesse paigaldust ei toimu, need on lisatud joonisele 3 vaid selleks, et arhitektuur peegeldaks reaalset süsteemi.

Arendusmeeskond lansseerib versiooniuuendusi kord nädalas, välja arvatud kriitiliste vigade ilmnemisel, kui seda tehakse jooksvalt. Iga nädalase süsteemi täispaigaldusega tekib süsteemis keskmiselt 30 viga.

Kogu rakendus on ühes Visual Studio .sln failis, jaotatud ära 130 projekti vahel, kokku umbes 130 000 rida C# koodi. Arendusmeeskond hoiab lähtekoodi versioonihaldussüsteemis Git.

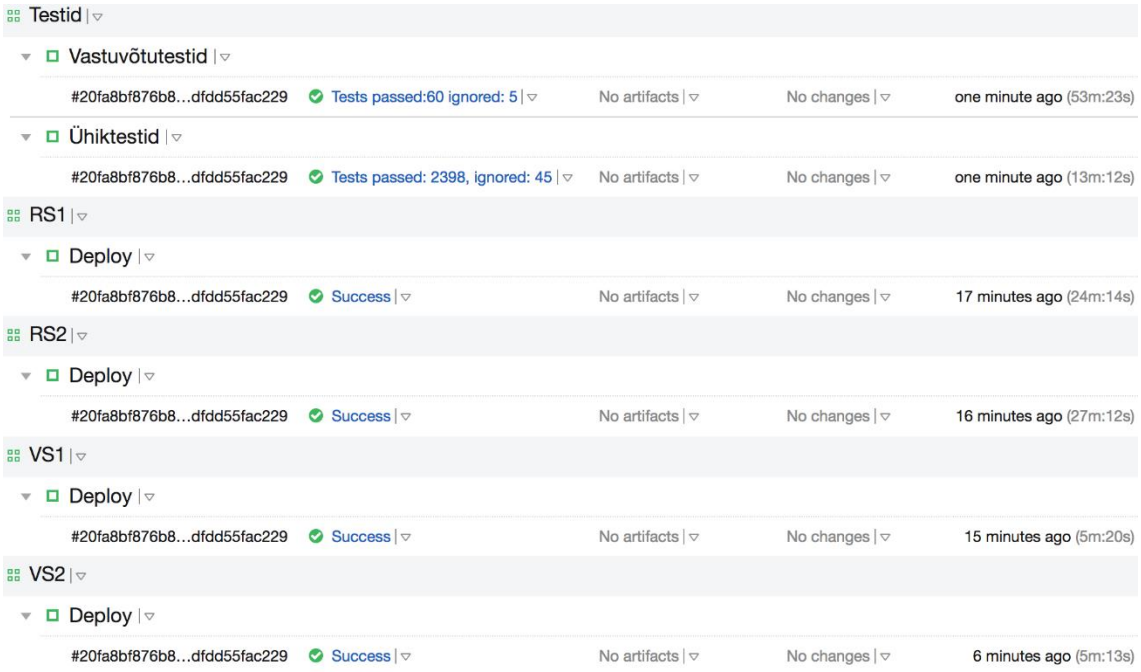
Koodimuudatuse integreerimisprotsess on näidissüsteemis üsna kaootiline. Kui koodimuudatus jõuab versioonihaldusesse, käivitatakse automaatselt ühiktestid. Nende tulemus on lihtsalt infoks ning ei piira kuidagi protsessi edasist kulgu. Arendusmeeskonnal (kuhu kuuluvad ka testijad) on võimalik käsitsi käivitada

vastuvõtuteste ning ka siinkohal tulemus ei piira edasist protsessi kulgu. Manuaalseks testimiseks käivitab arendusmeeskond ehitamise ning paigaldamise testkeskkonda. Kui seal ollakse tulemusega rahul, siis võetakse sama koodiversioon ning ehitatakse uuesti ning paigaldatakse toodangukeskkonda (vaata Joonis 4).



Joonis 4. Koodimuudatuse integreerimisprotsess näidissüsteemis.

Kekseks pideva integratsiooni serveriks on kasutusel TeamCity. Allpool on näha paigalduse protsesside jaotus TeamCitys (vaata Joonis 5).



Joonis 5. Paigalduse protsesside jaotus.

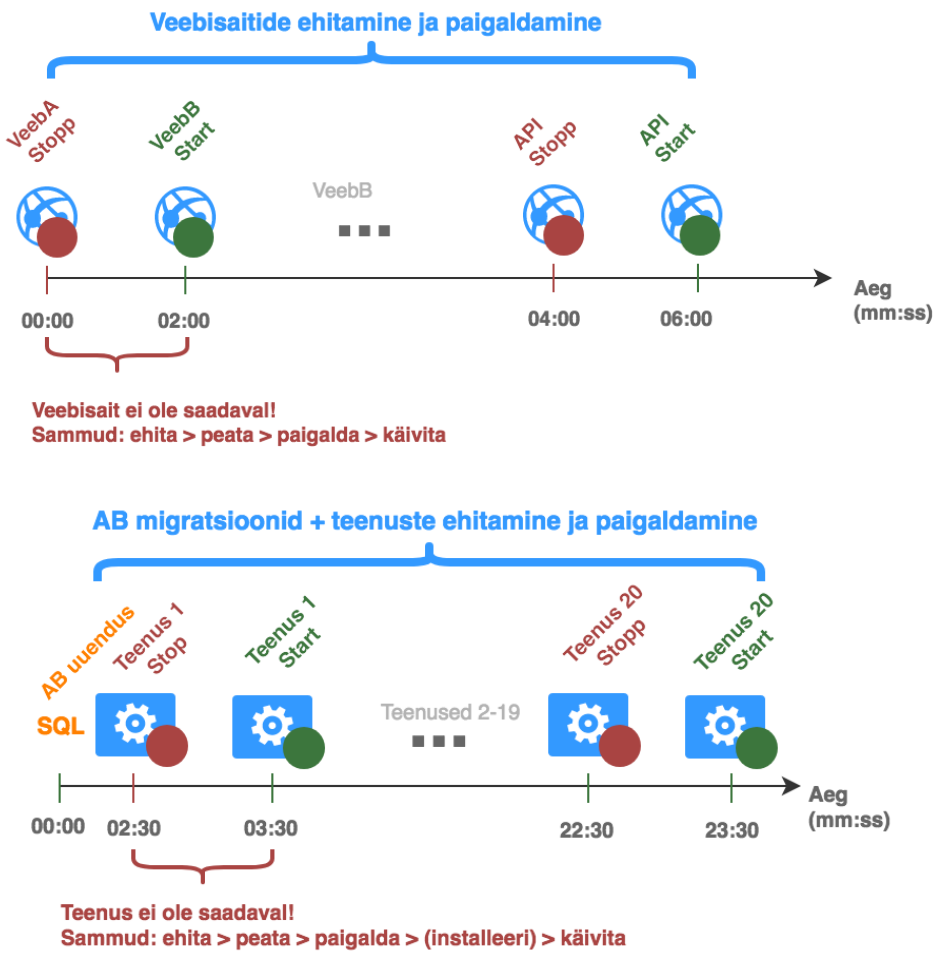
Jooniselt on näha, et rakendusel on 60 vastuvõtutesti, millede käitamine vältab üle 50 minuti. Lähemalt vaadates selgub, et kasutatakse Selenium teste ning Firefox brauserit. Lisaks on üle 2000 ühiktesti, millede läbimine võtab aega üle 13 minuti ning testimisraamistikest on kasutusel nii NUnit kui ka MSpec. Testid käivituvad automaatselt uue koodimuudatuse versioonihaldusest leidmisel ning nende läbimine või mitteläbimine ei piira paigaldamist. Järgnevad paigaldamise sektsioonid, mis on jaotatud masina kaupa. Paigaldamine antud kontekstis on veebiserveri puhul järgmine: ehitamine, peatamine, uuendamine ja veebisaitide käivitamine. Kokku vältab protsess üle 5 minuti. Teenuste puhul algab protsess ehitamisega, seejärel peatatakse vana teenus, kustutatakse vanad failid, kopeeritakse uued ning installeeritakse teenus. Kui teenus on uus, muudetakse keskkonnapõhiselt konfiguratsioon ning käivitatakse vastav teenus. Protsess vältab kokku üle 16 minuti. Nii ehitamiseks kui ka paigaldamiseks kasutatakse MSBuildi ning kogukonna poolt arendatud lisamoodulite kompleti MSBuild Community Tasks.

Etteruttavalt olgu öeldud, et antud infosüsteemis on pideva integreerimise ja paigaldamisega palju probleeme. Järgmistes osades räägib autor lähemalt pideva integreerimise ja tarne olemusest ning toob seejärel näidisinfosüsteemi põhjal parandusettepanekud.

5.1 Analüüs

Rakenduse ehitamine (*build*) ja paigaldamine (*deploy*) on mõlemad realiseeritud XML (*Extensible Markup Language*, laiendatav märgistuskeel) põhise MSBuild skriptide abil. Baasversioonis olevale funktsionaalsusele lisaks kasutatakse ka kogukonna poolt loodud lisamooduleid, mis eeldab, et moodulid on eelnevalt mõne administraatori poolt sinna paigaldatud. MSBuild on küll vana ning usaldusväärne rakendus, kuid on autori arvates XML-i tõttu liiga “jutukas”. Samuti on skripte keeruline siluda, kuna puudub mugav võimalus erinevate taimerite ja logimise kasutamiseks.

Mikrotasandil hälbeid peaaegu et ei märkagi, kuid kui vaadata kogu protsessi tervikuna, on kitsaskohad selgelt näha. Allpool on näidatud veebisaitide ja teenuste ehitamise ning paigaldamise protsess ajajoonel (vaata Joonis 6).

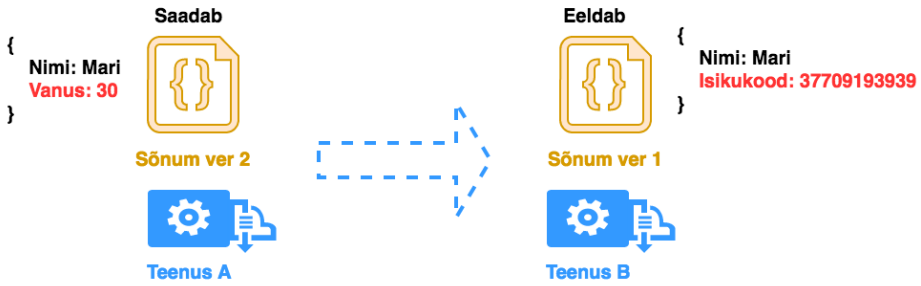


Joonis 6. Veebisaitide ning teenuste ehitamine ja paigaldamine.

Järgnevalt analüüsitakse ülemist joonist, ehk veebisaitide ehitamist ja paigaldamist. Antud protsess käivitatakse käsitsi mõlemas veebisaite majutatavas masinas (VS1 ja VS2) korraga. Esiteks, juba see on väär, et paigaldust läbiviiv isik peab käivitama mitu protsessi. Veebisaidid uuendatakse ükshaaval, alustatakse peatamisega, järgneb ehitamine ja käivitamine ning seejärel liigutakse järgmise veebisaidi juurde. Probleem on selles, et kõik veebisaidid (ja ka teenused) sõltuvad ühest relatsioonilisest andmebaasist, mille uuendamist alustatakse veebisaitidega paralleelselt (näha teenuste ajajoonel). Teenuste ajajoonelt on näha, et andmebaasi uuendus võtab aega 2,5 minutit, mis tähendab, et kui veeb A käivitub, siis on veel kasutusel vana andmebaasi struktuur ning on oht vigade tekkimisele. Kui andmebaas on uuendatud, on töös veel vana versiooniga API (*Application programming interface,* rakendusliides), mis eeldab süsteemilt vana andmebaasi struktuuri ning jällegi on oht vigade tekkimiseks. Paigalduse

protsess oleks oluliselt kiirem, kui sellest jätta välja ehitamine, ehk kasutada juba eelnevalt ehitatud tehiseid.

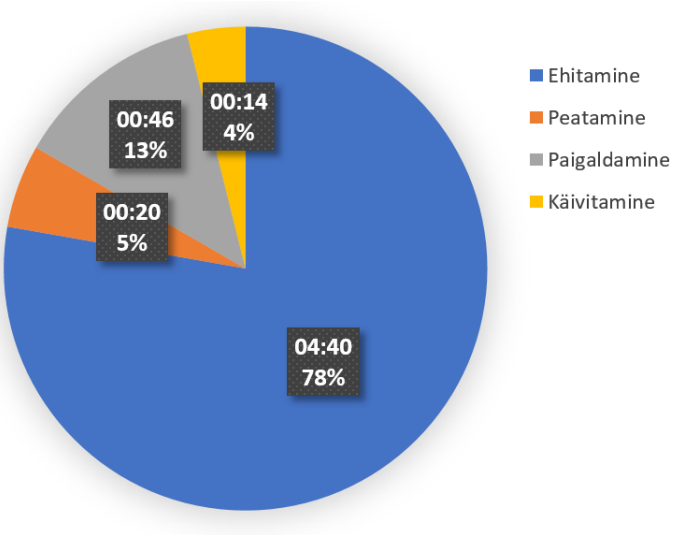
Järgmiseks analüüsitakse teenuste paigaldamise protsessi. Skeemilt on puudu teenuste esmakordne installeerimine, kuna seda teostatakse üsna harva. Jooniselt on näha, et kõik tegevused on teostatud järjestikku: alustatakse andmebaasi uuendamisega ning seejärel värskendatakse kõik teenused. Oluline on mainida, et kõik teenused töötavad nii kaua, kuni uuendamisjärg nendeni jõuab, ning pärast vastava osa uuendamist teenus jälle käivitatakse. Sarnaselt eelmise lõiguga on ka siin protsessi kõige suurem puudus see, et rakendus töötab terve paigaldamistsükli vältel puudulikult: osad teenused omavad erinevatest moodulitest uuemat versiooni kui teised. Kui nüüd teenus B eeldab, et teenus A saadab talle sõnumi struktuuriga versioon 1, siis kahe mooduli suhtlus on häiritud kui teenuses A on sõnumi versiooniks juba 2 (vaata Joonis 7).



Joonis 7. Teenuste vaheline sõnumi saatmine.

Suureks probleemiks on ka andmebaasi uuendused, kuna kõikidel süsteemi osadel on sõltuvus kesksest andmebaasist ehk kõik süsteemi osad eeldavad kindlat andmebaasi struktuuri. Kui andmebaasiuuendusega kaotatakse ära mõni tabel või selle väli, saavad kõik veel uuendamata teenused selle tabeli poole pöördumisel vea.

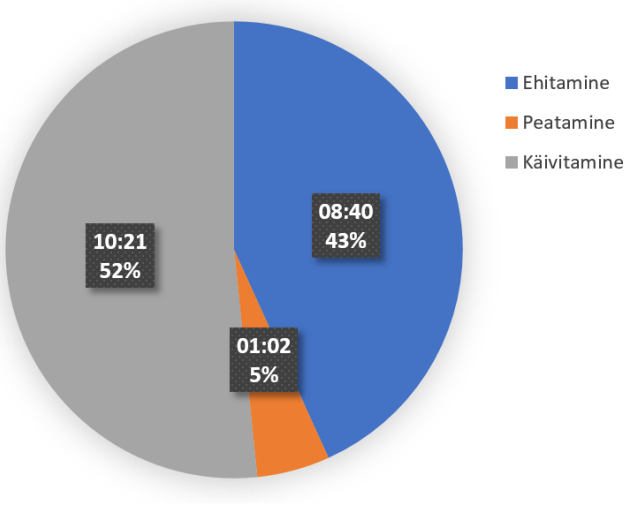
Allpool on veebisaitide ehitamiseks ja paigaldamiseks kulunud aeg kumulatiivselt (vaata Joonis 8).



Joonis 8. Veebisaitide ajajaotus.

Jooniselt järeldub, et 78% ajast ehitatakse veebisaidi tihendatud pakki ning ainult 22% ajast kulub reaalsele paigaldusele (saidi peatamiseks, paigaldamiseks ja käivitamiseks). Kui siit kõrvaldada ehitamise protsess, on võimalik paigaldamine oluliselt kiiremaks teha.

Järgmisena on kujutatud graafiliselt teenuste paigaldamisega seotud protsess (vaata Joonis 9).



Joonis 9. Teenuste ajajaotus.

Diagrammilt on välja jäetud paigaldamise (failide kustutamine, kopeerimine) osa kuna see on marginaalne, sest ehitamise väljundkataloogiks on märgitud kaust, kus paigaldatud teenus paikneb ja vanade failide kustutamine ning konfiguratsiooni muutmine ei võta

praktiliselt üldse aega. Kui veebisaitide puhul läks enamus aega ehitamise peale, siis teenuste puhul kulub enamus aega nende käivitamisele.

Vaadates ehituse ja paigalduse protsessis osalevate serverite koormust, on selgelt näha, et süsteemi ressursid on praktiliselt kasutamata: protsessori 8 tuumast teevad tööd vaid 2 ja seda ka keskmiselt 60% ulatuses. Failisüsteemi taga on kiired SSD (*Solid State Disk*) kettad RAID10 seades ning ka selle jõudluse piirist jäi ehituse ja paigaldus oma haripunktis kaugele.

Võttes arvesse eelpool väljatoodut, on võimalik järgnevate parendustega muuta protsessi efektiivsemaks (vaata Tabel 1).

Tabel 1. Parendusettepanekud.

Id	Kirjeldus	Kasu
P1	Lahutada ehitamise ja paigaldamise protsess	Aeg ~ -50%
P2	Viia võimalikult palju tegevusi paralleelseks, kasutades ära nii palju süsteemi jõudlust kui võimalik	Aeg ~ -50%
P3	Süsteemiosade üheaegne seiskamine, uuendamine ja käivitamine	Töökindlus
P4	Paigalduse protsess ei tohi süsteemis põhjustada vigu	Töökindlus
P5	Paigaldamine on võimalik vaid juhul, kui eelnevalt on edukalt läbitud testimine	Töökindlus
P6	Paigaldamine ühe nupuvajutusega	Väiksem tõenäosus eksimiseks

Kuna ehitamine ja paigaldamine on plaanis lahku lüüa ning esimene ka optimeerida kiiruse peale, võib see protsess hõivata üsna suure osa protsessori koormusest. Seega tuleb üles seada eraldiseisev server ehitamise eesmärgil. Samuti tuleb luua võimalus ehitatud tehiste efektiivseks transpordiks serveritesse, kuhu need paigaldada tuleb.

6 Pidev integreerimine

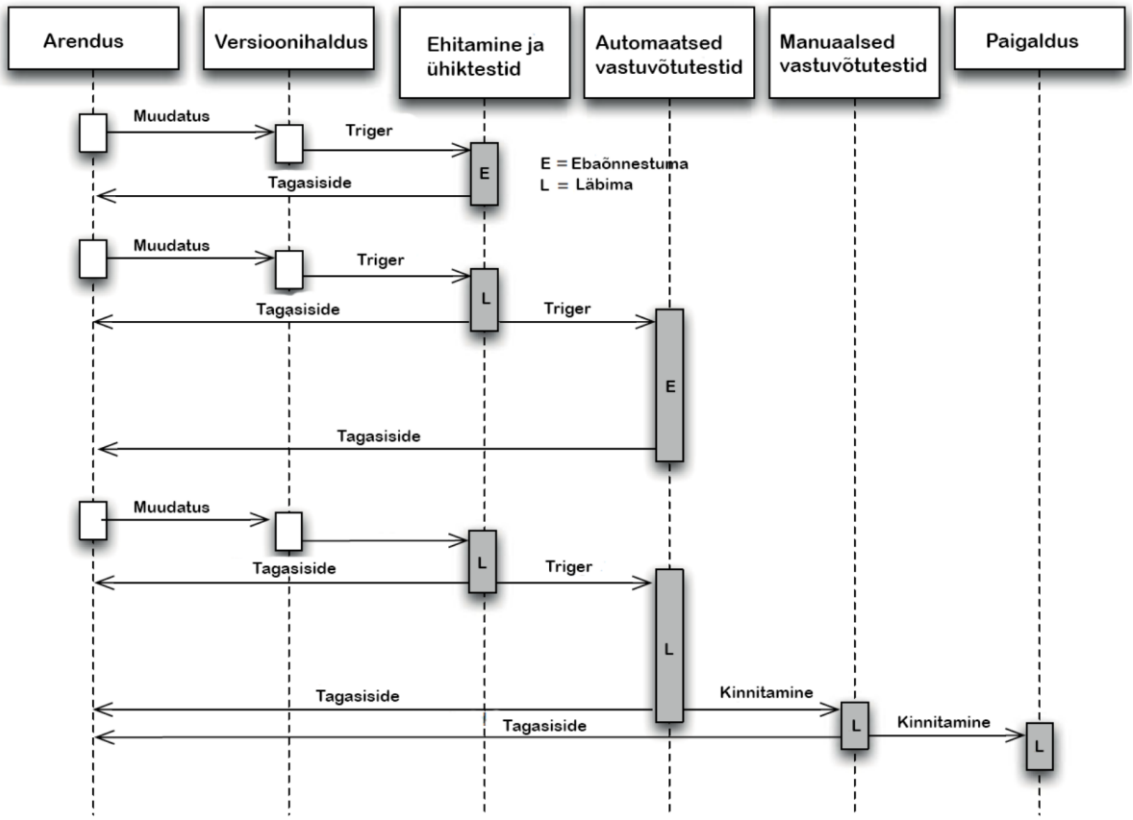
Pidev integreerimine (*Continuous Integration*) on protsess, milles arendajad tarnivad järjepidevalt tarkvara muudatusi. See on tavaliselt automatiseeritud protsess, mille käigus DevOps (vaata peatükk 3) meeskonnad aitavad projekti terve eluea jooksul hoida kiire ja paindlikuna. Sellise süsteemi eelduseks on mingi tarkvaratoote võime kiirelt kohaneda turul toimuvale. Kui klientidel tekib vajadus kohandatud toote järele, siis ei tule enam oodata pikka protsessi muudatuse tarnimiseks. Selle tsükli lühenemine on ka oluline veaolukordades, kus turvaauk või kehvalt tehtud tarkvara võib väga kalliks maksma minna, eriti juhul, kui seda ei suudeta õigel ajal parandada. Pidev tarne on moodsa tarkvaraarendusmaailma üks tugisammastest. Arendusvahendid on juba piisavalt head ning klientide vajadused on järjest keerulisemad. Pidev tarne ja paigaldamine on juba pigem standard kui erand [8]. Kent Beck mainis pidevat integreerimist juba aastal 1999 oma raamatus Extreme Programming Explained. Nüüd aastatid hiljem on tekkinud palju erinevaid lähenemisi ja tööriistu, mis on küpsed kasutamiseks ka üsna konservatiivsetes projektides.

Tarkvara paindliku arendamise puhul puudub põhjalik eelnev analüüs ning toimub pidev katsetamine (eeldusel, et testid pakuvad selleks piisavat kindlust). Paraku vähegi keerulisema infosüsteemi puhul ei suuda inimene kõiki muudatuse tagajärgi ette näha ning selleks on vaja efektiivset protsessi, mis annaks arendajale ning äripoolele võimalikult kiiret tagasisidet. Kuigi pideva integreerimise tarkvara on kasutamiseks piisavalt küps ning pakub palju võimalusi, nõuab see meeskonnalt siiski teataval määral distsipliini. Oluline on tekkinud vigadele reageerida kiirelt, vastutavat isikut peaks olema võimalik teavitada veast automaatselt. Näitena võib siinkohal tuua ühiktestides vea tekkimisel koodimuudatuse autorile automaatse emaili saatmist. Tagasiside tsükkel on eelistatavalt nii lühike kui võimalik ning eeldab protsesside optimeerimist ja seiret, eriti töövoo alguses olevate automaatsete protsesside osas (nt ehitamine või ühiktestide käitamine). Tuleks seada üles protsessid, mis annavad häiret, kui mõni protsess võtab oodatust kauem aega, näiteks ehitamine kestab rohkem kui 3 minutit vms.

Terve protsess võiks olla nii automatiseeritud kui võimalik, et vältida inimfaktorist tulenevaid vigu. Mõõtmaks protsesside efektiivsust saab üles seada erinevaid mõõdikuid, näiteks ehitamiste arv päevas, keskmine ehitamise aeg, keskmine ühiktestide kestus,

ebaõnnestunud testide arv, paigaldamise aeg, koodi katvus testidega, terve tsükli pikkus jne. Mõõdikute peale saab ehitada automaatse seire, mille abil saaks teavitada vastutavat isikut olukorra halvenemisest. Äärmuslikemal juhtudel võib isegi automaatse protsessina keelata uue koodi lansseerimise, et hoida ära olukorra eskaleerumist.

Töövoog (*pipeline, build chain*) on jada kindlas järjestuses olevaid tegevusi, mille läbimisel ehitatakse, testitakse ja paigaldatakse uus tarkvara redaktsioon. Olenevalt toote keerukusest ning protsesside küpsusest, võib kogu protsess olla automaatne või poolautomaatne, mis enamasti tähendab lõpufaasis manuaalset testimist ning paigaldamist (vaata Joonis 10) [7, lk 109].



Joonis 10. Pideva integreerimise ja paigaldamise töövoog [7, lk 109].

Koodis tehtud muudatused peavad läbima igat sammu, vea ilmnemisel liigub protsess tagasi algusesse ehk arendusse. Olenevalt projektist ja protsessidest võib manuaalse testimise ka ära jätta ning seejärel on võimalik kogu töövoogu läbida automaatselt. Manuaalse testimise puhul ei ole sageli otstarbekas testida igat muudatust eraldi, vaid näiteks arendustsükli jooksul valmis saanud funktsionaalsus tervikuna. Viimane variant

võib muidugi vea korral muuta defektse koha tuvastamise keerulisemaks. Töövoog lõppeb paigaldusega soovitud keskkonda, milleks võib olla mõni toodangulaadsetest testkeskkondadest või toodang ise. Infosüsteemide keerukuse tõttu on siiski võimalik, et rakenduse testitud redaktsioonis on viga, mida töövoo läbimisel ei suudetud tuvastada, ning mis ilmneb alles pärast paigaldamist. Sellise olukorra peab tuvastama seire ning vastavalt olukorrale kas paigatakse tekkinud viga või tehakse tagasipööre eelmisele versioonile (*rollback*). Mõlemal juhul tuleb töövoogu täiustada, et sama tüüpi viga suudetaks järgmisel korral enne paigaldamist avastada.

Pideva integreerimise teeb mugavaks hea tarkvara, mida on turul saadaval üsna palju. Leidub nii tasulisi kui ka tasuta variante, kasutamiseks pilves või majutamaks enda serveris. Tuntuimad neist on Jenkins, TeamCity, Bamboo, GitLab, AppVeyor jt. Eelpool mainitud tarkvarad on pideva integreerimise juhtsüsteemideks, mis koordineerivad koodimuudatust läbi töövoo. Osa tarkvarasid katavad rohkem elutsükleid kui teised. Näiteks TeamCity on oma olemuselt mõeldud ehitamiseks ja testide käitamiseks, kuid GitLab on võtnud laiema suuna. Viimane pakub lisaks ka projekti- ja koodihaldustarkvara, kuid põhifookuseks on GitLabil siiski koodihaldus. Sageli kasutavad ettevõtted projektihaldustarkvara, millel ei ole pideva integreerimise funktsionaalsust (näiteks Asana). Sel juhul tuleb ettevõttel vastu võtta otsus, kas minna üle tervikplatvormile või liidestada projektijuhtimis- ja pideva integreerimise tarkvara. Õnneks on tänapäeval enamik tarkvaral olemas rakendusliides ehk API, mis liidestamist üsna edukalt võimaldab.

Kuna näidissüsteem kasutab pideva integratsiooni serverina TeamCityt, siis tutvustab autor seda tarkvara lähemalt. Rohkem kui 10 aastat turul olnud, on TeamCity tõestanud end kui töökindla ja rikkaliku funktsionaalsusega tarkvarana. TeamCity on keskendunud koodi ehitamisele ja testimisele, kuid sellega on võimalik täiesti edukalt ka läbi viia erineva keerukusega paigaldusi. Versioonihaldustarkvaradest (VCS) toetab see kõiki tuntumaid: ClearCase, CVS, Git, Mercurial, Perforce, StarTeam, Subversion, Team Foundation Server, SourceGear Vault ja Visual SourceSafe.

Tasuta ehk Professional versioon piirab kasutamist 20 konfiguratsiooni ja 3 ehitamisagendini, mis on autori arvates keskmise suurusega ettevõttes targalt organiseerides täiesti piisav. Ehitamisagente on võimalik jooksvalt juurde osta ning nõudlikumal kasutajal on võimalik soetada Enterprise versioon, mis annab rohkem

konfiguratsioone ning parema kasutajatoe. TeamCity on platvormist sõltumatu, ehk seda on võimalik paigaldada nii Windowsile, Linuxile kui ka MacOS platvormile. TeamCity kasutab agendipõhist lähenemist, ehk ehitamist viivad läbi eraldiseisvad teenused ehk agendid, kes saadavad andmeid kesksele serverile. Selline lähenemine võimaldab lihtsalt käivitada ehitamisi erinevatel platvormidel.

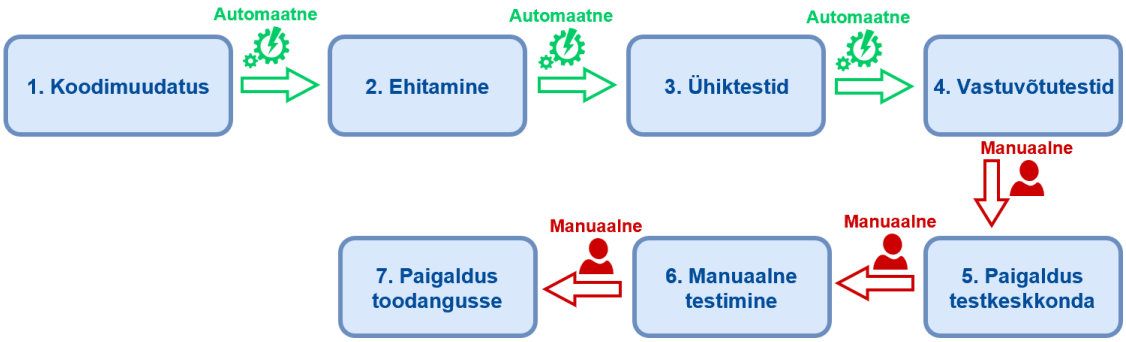
TeamCity on eelkõige mõeldud ehitamiseks ja testide käitamiseks, kuid saab tänu võimalusele erinevaid skripte käitada hakkama ka paigaldamisega (sellest lähemalt peatükis 10). Erinevate pistikprogrammide abil on võimalik liidestada TeamCityt ka erinevate projektihaldustarkvaradega (näiteks Jira, Bugzilla või YouTrack).

TeamCity projektid saab kirjeldada läbi veebiliidese ning vajadusel hoida muudatusi ka eraldi versioonihalduses. Kellele läbi veebiliidese kirjeldamine ei sobi, võib seda teha selleks loodud spetsiaalkeele Kotlin abil [14].

Igale projektile on võimalik seadistada eraldi agent, kes hakkab vastavat tööd teostama. Kui seda ei tehta, siis TeamCity võtab fondist ühe suvalise vaba agendi. Kindlasti tuleks seadistada TeamCity nii, et toodangu serverites olevad agendid ei oleks vaikimisi fondis kättesaadavad, vastasel juhul võib see häirida tööd toodangu masinas.

TeamCityl on sisse ehitatud kasutajate ja õiguste haldus. Kasutajate halduse saab siduda Windowsi domeeniga või LDAP-iga (*Lightweight Directory Access Protocol*, kataloogipöörduse kergprotokoll). Igast muudatusest jääb maha jälg, mis teeb auditeerimise võimalikuks [15]. Õigusi on mõistlik jagada keskkonna põhiselt ehk testkeskkondades võivad arendajad peaasjalikult kõike teha, toodangu keskkondades võiks olla vaikimisi kõigil vaatamise õigus ning muutmise õigus võiks jääda ülalhoiu meeskonnale, kes vastutab sinna paigalduse eest.

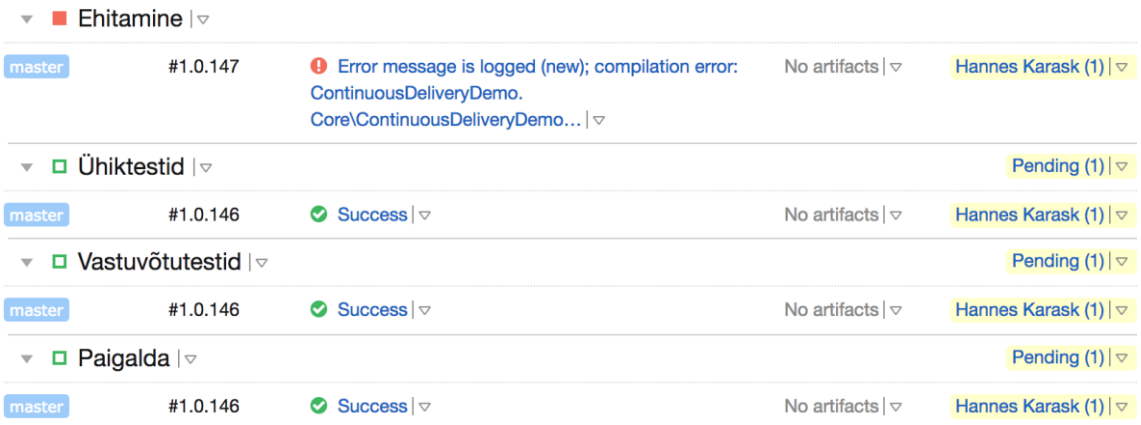
Lähtuvalt eespool kirjeldatud pideva integratsiooni protsessile, saame näidisprojektis oleva koodimuudatuse protsessi ümber modelleerida järgmiselt (vaata Joonis 11).



Joonis 11. Uuendatud koodimuudatuse integreerimise protsess.

Muudatuse protsessi töövoog on nüüd lineaarne ning selgelt defineeritud. Koodimuudatus peab läbima esmalt ehitamise, enne kui see on valmis järgmisteks etappideks. Ainult pärast edukat testimist on võimalik muudatus paigaldada testkeskkonda, kust see hiljem edasi toodangusse jõuab.

Seoses uuenenud protsessiga, organiseerime järgnevalt ümber ka TeamCitys olevad protsessid (vaata Joonis 12).



Joonis 12. Pideva integreerimise töövoog TeamCitys.

Võrreldes algse protsessiga on muudatusi üsna palju. Tegevused ei jaotu enam masina nimede järgi, vaid protsessi olemuse järgi. Masina nimed on tegelikult ju seadistuse osa, mitte protsessi osa. Paigaldamise ja ehitamise samm on viidud lahku ning kogu töövoog on lineaarne. Töövoog on esitatud vertikaalselt ning saab alguse kõige ülemisest sammust „Ehitamine“, mis on punane, kuna kompileerimisel tekkis viga. Järgmised sammud on rohelised, mis tähendab, et need on õnnestunud. Kui mingil sammul tekib viga, ei saa erinevalt näidissüsteemist minna edasi järgmisele, kus ühikteste oli võimalik käitada

ülejäänud protsessist sõltumatult. Oluline on märkida, et TeamCitys tuleb määrata eraldi ära, et allavoolu olevad projektid päriksid versiooninumbrid (numbri kujust lähemalt jaotises 7.2) ülesvoolu olevatelt projektidelt. Kõik sammud enne paigaldamist on automaatsed, ehk kui TeamCity saab signaali uuest koodimuudatusest (kas VCS on seadistatud esitama muudatusi või TeamCity kontrollib seda etteantud intervalliga), alustatakse ehitamise sammuga. Ühiktestid ja vastuvõtutestid käivituvad automaatselt kohe kui eelmine samm on edukalt lõppenud.

6.1 Integratsioonid

Tarkvara arenduse protsessis on tähtis roll toetataval tarkvaral, näiteks projekti juhtimiseks kasutatavad süsteemid: Asana, Jira, YouTrack jt. Oluline on võimalikult palju infot koguda ühte kohta, et protsessist oleks hea ülevaade. Kui arendaja on mõne ülesande lõpetanud ning oma koodimuudatused kesksesse versioonihaldusesse pannud, peaks olema selle kohta märge ka projektihaldustarkvaras vastava ülesande ajaloos Nii teab testija, et arendaja poolt on töö üle antud ning tööjärg on tema käes.

Projektijuhtimise jaoks kasutatava tarkvara kõrval on olulised ka erinevad jutuside (*chat*) tüüpi tarkvarad, näiteks Skype või Slack. Isikud on grupeerunud huvigruppideks ning saavad erinevaid teateid. Suhtlus on nii kõige vahetum ja pigem väheformaalne. Erinevad tarkvarad saavad gruppidesse saata sõnumeid, näiteks millist keskkonda millise versiooniga uuendatakse või millised testid ei läbinud edukalt. Nii on kõigil huvilistel operatiivne info süsteemis toimuvast. Seire käigus leitud vead peaksid jõudma õigete inimesteni. SMS-i ja emaili kõrval on mõistlik kasutada ka jutusidet, sest enamikul meist on taskus nutiseadmed, mis suudavad reaalajas teateid vastu võtta.

Uue tarkvara versiooni lansseerimisel on enamasti vaja ka huvipooltele esitada väljalasketeade (*release notes*). Viimast on võimalik automatiseerida kui kasutada projektihaldustarkvara, mis võimaldab genereerida nimekirja kõigist redaktsiooni jõudnud muudatustest.

7 Ehitamine

Ehitamine on pideva integreerimise töövoos esimene samm, milles lähtekoodist luuakse masinale optimeeritud ja sageli kahendkoodis esitatud, tehised (*artifact*). See samm peaks käivituma automaatselt, kui keegi lansseerib koodimuudatusi. .NET maailmas esitatakse lähtekood näiteks .cs failidena (keeleks C#), mis on omakorda organiseeritud projektidesse (laiendiga .csproj) ning projekti ehitamisel tekib DLL (*dynamic link-library*). Projekte võib suuremas süsteemis olla sadu ning nad koonduvad omakorda Solution faili (.sln) alla.

.NET rakenduse ehitamiseks on standardiks kujunenud Ant stiilis XML põhine MSBuild. Enamik skriptidest on automaatselt IDE (*Integrated development environment*, arenduskeskkond), mis antud juhul on Visual Studio, poolt loodud ning nende käsitsi kohaldamist sageli ette ei tule. MSBuild on käsureapõhine tööriist, millel on üsna palju erinevaid valikuid ning millest sõltub olulisel määral ehitamise kiirus. Ehitamise kiirus on oluline vähendamaks tagasiside tsüklit. Mida kiirem on ehitamine, seda kiiremini saab arendaja tagasisidet selle kohta, et kas rakendus läbib esimese kontrolli. Mõnel pool levib praktika teha öiseid ehitamisi (*nightly builds*) just sellel põhjusel, et ehitamine ja testimine on aeglased [7, lk 65]. Pidev integreerimine ja tarne oma olemuselt välistavad selle praktika. Edasi uurib autor kuidas protsessi muuta nii kiireks kui võimalik, et seda saaks teha igal ajal ja tihti.

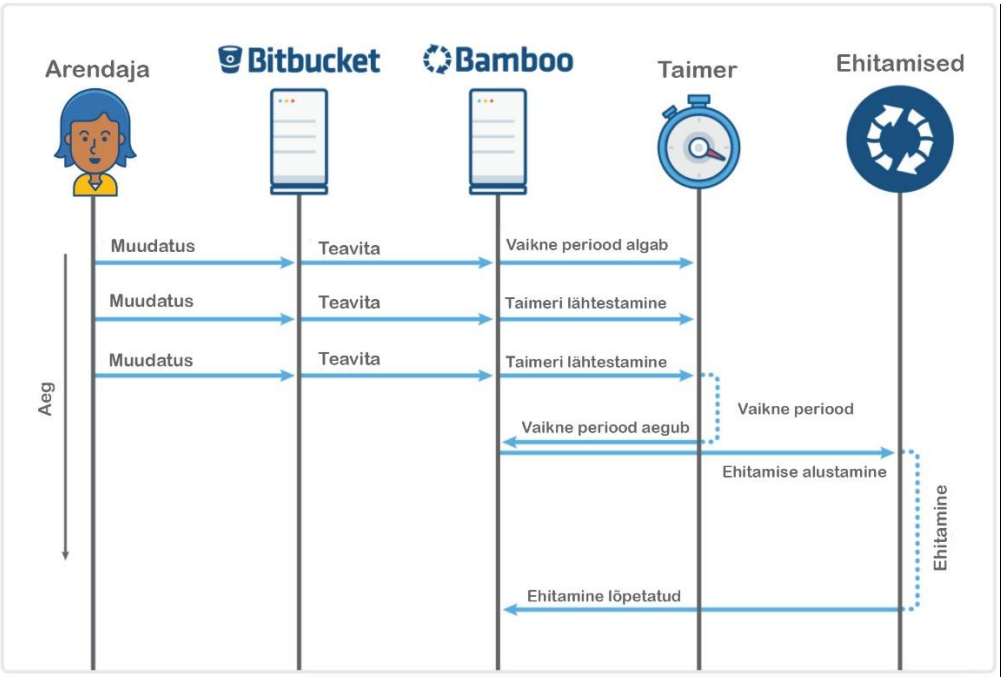
Ehitusserveri üles seadmisel tuleb sinna paigaldada SDK selle raamistiku versiooniga, mis on ehitatavas tarkvaras kasutusel. Samuti tuleb paigaldada Microsoft Build tools ning veebisaitide ehitamisel kas kopeerida selle spetsiifilised kaustad masinast, kus on Visual Studio installeeritud, või viidata NuGet pakettidele. MSBuildi kasutamisel ehitusserveris tuleks seadistuses ära märkida kindel MSBuildi versioon, mis võiks olla kõige vanem, mis meeskonnas oleva arendaja kasutada on. Vastasel juhul kui keegi meeskonnast kasutab näiteks C# v6 spetsiifilist süntaksi, siis ehitusserver laseb selle läbi, kuid ainult C# v5 toega arendaja masin jääb siinkohal hätta.

MSBuild on TeamCitysse sisse ehitatud ning parameetreid saab ette anda otse ehitamiskonfiguratsioonis või kirjeldada ära projektipõhiselt (*system* prefiksiga). Viimast varianti autor ei soovita, kuna need võivad ununeda ning uue ehitamiskonfiguratsiooni loomisel võib ehitamisprotsess käituda ettearvamatult.

On tavaline, et ehitamise käigus tekib väljundisse mitmeid hoiatusi (*warning*), mis enamasti ei mõjuta olulisel määral rakenduse tööd. Võib ette tulla juhtumeid, kus see aga mõjutab, näiteks kui ei leita kindla versiooniga sihtraamistikku. Sel juhul võtab kompilaator suvalise selles masinas leiduva raamistiku ning rakenduse käitamise ajal võivad tekkida probleemid. Uue projekti alustamisel oleks mõislik hoida hoiatuste nimekirja puhtana, vanema projekti puhul võib kasutada positiivset registrit, kus on kirjas lubatud hoiatused.

Ehitamise protsessi võib läbi kukutada ka juhul, kui mõned karakteristikud ei vasta enam piirväärtustele, näiteks ehitamise aeg on läinud liiga pikaks või hoiatuste arv on eelmise ehitamisega võrreldes tõusnud.

Pideva integreerimise (edaspidi PI) serveris on mõistlik seadistada ehitamise alustamise triger nii, et see käivituks pärast iga koodimuudatust versioonihalduses. Viimane on vajalik selleks, et täpsemalt määrata vigane muudatus (ja muudatuse tegija) ning selle tulemusel kiiremalt viga likvideerida. Seda saab teha vaid eeldusel, et ehitamine ei võta liiga kaua aega ning leidub piisavalt arvutusressurssi. Kui viimane on aga piiratud, saab kombineerida mitu lähestikku asetsevat koodimuudatust ühte ehitamisse, märkides ära aja (näiteks 30 sekundit), kui kaua ootab PI server enne ehitama hakkamist veel muudatusi (vaata Joonis 13).



Joonis 13. Pideva integreerimise tarkvara töövoog [16].

Joonisel 13 on näha kolme koodimuudatust, mis saadetakse arendaja poolt VCS süsteemi Bitbucket. Viimane teavitab muudatusest PI serverit Bamboo, mille peale käivitatakse taimer ja algab vaikne periood. Iga järgnev koodimuudatus lähtestab taimeri, kui jõuab seadistatud välte piiridesse, vastasel korral alustab PI server ehitamisega ning muudatus lisatakse järgmisesse tsüklisse.

7.1 Kiiruse optimeerimine

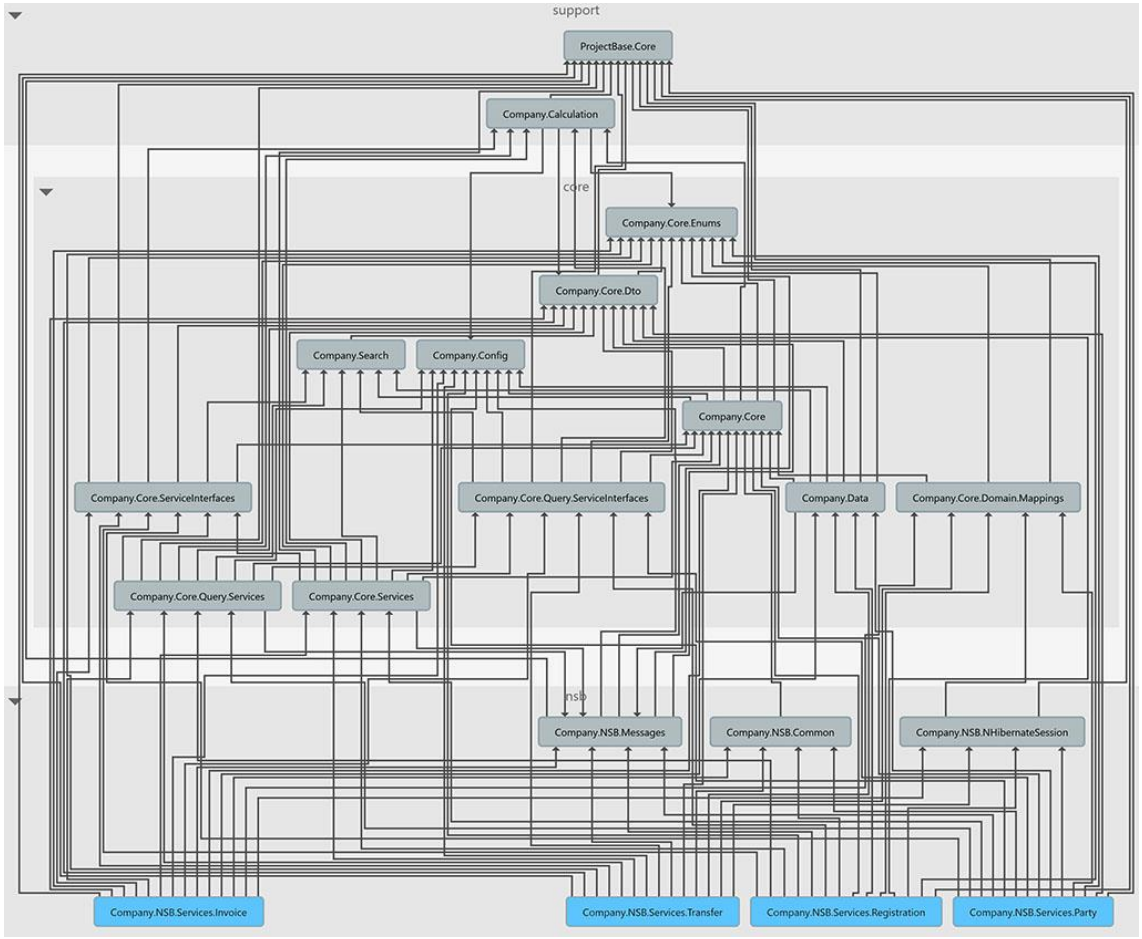
Enne ehitamise alustamist tuleks puhastada eelmise ehitamise tehised. MSBuildile saab ette anda sihiks *Rebuild*, mis puhastab eelmise ehitamise tehised ning seejärel ehitab uued. Ehitamine toimub sageli verisoonihalduses oleva lähtekoodiga samas kaustas ning sel juhul on võimalik kasutada ka versioonihalduse tööriistu puhastamaks faile, mis lähtekoodi hulka ei kuulu. Järgnev näide on toodud Giti põhjal:

```
git clean -x -d -e .vs -e packages -e *.csproj.user -e *.suo –f
```

Antud Giti käsk kustutab ära kõik failid ja kaustad, mida ei ole eraldi lubatud -e parameetriga. Oluline on jätta alles *packages* (milles on NuGeti moodulid) kaust, kuna seal sees võib olla sadu megabaite mooduleid, mille uuesti laadimine võib võtta üsna kaua aega.

Üks olulisemaid faktoreid ehitamiskiiruses on sammehitamine (*incremental build*), mille käigus MSBuild üritab leida seosed lähte- ja sihtfailide vahel. Need seosed võivad olla üks-ühele (ühele lähtefailile vastab üks sihtfail) või palju-ühele (mitu lähtefaili moodustavad ühe sihtfaili, nt mitu .cs lähtefaili moodustavad ühe DLL-i). MSBuild nimelt võrdleb failide ajatempleid: kui sihtfailil on vanem tempel, siis luuakse fail uuesti, vastasel korral jäetakse see puutumata [17].

Ehkki sammehitamine toimib vaikimisi Visual Studio genereeritud projektidega, võib käsitsi keerulisemaid MSBuilde skripte luues selle mehhanismi ära lõhkuda. Autor tegi katse, milles andis MSBuildile ette 50 projekti, mis omakorda kasutavad umbes sama palju ühiseid projekte ning mõõtis aega, mis kulub ehitamiseks sammehitamisega ja ilma. Joonisel 14 on näha nelja suvaliselt valitud (märgitud sinisega) projekti sõltuvused.



Joonis 14. Väljavõte projektide sõltuvusdiagrammist.

Ilma sammehitamiseta kulus ehitamiseks keskmiselt 3,8 ja sammehitamisega kõigest 0,5 minutit.

Aastal 2010 tuli välja 1.0 versioon .NET maailma paketihaldurist nimega NuGet [18]. See võimaldab rakenduses mugavalt kasutada kolmanda poole loodud komponente. Kui eelnevalt oli vaja mahukaid binaarfaile hoida koos rakendusega, siis nüüd piisab viitest paketile ning rakenduse ehitamisel laetakse vastav komponent alla. Suuremate süsteemide ehitamisel mõjub NuGet pakettide olemasolu kontroll ehitamise kiirusele halvasti. Pakettide taastamine tuleks teha enne ehitamist Nuget käsurea utiliidi abil. Ehitamise ajaks tuleb öelda MSBuildile, et ei ole vaja kontrollide pakettide olemasolu. Selleks tuleb määrata parameetri *RestorePackages* väärtuseks *false*. Keelates ära pakettide taastamise, sai autor näidissüsteemis keskmiseks kiirusekasvuks 35%.

Analüüsi osas (vaata jaotis 5.1) jõuti järeldusele, et näidissüsteemis tuleb ehitamise osa lahku viia paigaldamisest ning pideva integreerimise peatükist saime selleks kinnituse

(vaata peatükk 6). Järgnevalt uurib autor, kuidas optimeerida näidissüsteemis ehitamise kiirust.

MSBuild kopeerib vaikimisi sihtkausta ka projektiga seotud xml ja pdb failid, mis mõjutavad samuti ehitamise aega. Määrates MSBuild parameetri AllowedReferenceRelatedFileExtensions väärtuseks mingi suvalise mitte eksisteeriva laiendi (tühja väärtust ei lubata), saame jällegi ehitamise aega vähema kopeerimise arvelt kiiremaks [19]. Selle parameetri määramine vähendas keskmiselt näidissüsteemis ehitamisele kulunud aega 23%.

Tänapäeval on enamik arvuteid mitmetuumalised ning paljud rakendused kasutavad seda ära, et tööd tuumade vahel jaotada. MSBuild kasutab vaikimisi ehitamiseks ühte protsessi, kuid seda on võimalik muuta parameetri *maxcpucount:<nr>* abil. Kui jätta protsesside arv parameetrist ära, tekitab MSBuild protsesse vastavalt tuumade arvule. Sisse peab olema lülitud ka valik *BuildInParallel*, mis on vaikimisi aktiivne [20]. Näidisprojekti ehitamisel paralleelselt 8-tuumalises serveris oli keskmine ajavõit 46%.

MSBuild kuvab vaikimisi väljundisse üsna palju infot, mis mõjub samuti kiirusele halvasti. Parameetri *verbosity* abil saab väljundisse saadetava sisu muuta minimaalseks, mis testsüsteemis tõi ehitamisel keskmiseks kiiruse kasvuks 15%. Järgnevalt on toodud erinevate parameetrite mõju ehitamisele (vaata Tabel 2).

Tabel 2. Ehitamise kiirust mõjutavad parameetrid.

	1	2	3	Keskmine (mm:ss)	Muut
Vaikimisi seaded	1:35	1:35	1:37	1:35	
Package restore off	1:05	1:01	1:01	1:02	35%
AllowedReferenceRelatedFileExtensions	0:51	0:47	0:47	0:48	23%
BuildInParallel	0:26	0:26	0:26	0:26	46%
Verbosity	0:22	0:23	0:23	0:22	15%
Kokku ajavõit			**77%**		

Tulemustest selgub, et kõige kõigu suurema ajavõidu annab ehitamisel paralleelsus ning kõikide parameetrite rakendamisel on ehitamise aega võimalik vähendada rohkem kui 70%. Suutsime näidisprojektis viia ehitamise ajalt 1:35 ajale 0:22, mis vähendab oluliselt tagasisideväldet koodi muudatuse ja ehitamise vahel.

7.2 Versioonitähistus

Tarkvara uuenedes muudetakse sellel versiooninumbrit. Avalikult levinud tarkvara puhul on üsna levinud semantiline tähistusviis (*Semantic Versioning*). Sellise viisi puhul koosneb versiooninumber põhi (*major*), väikesest (*minor*) ja parandus (patch) numbrist, mis on eraldatud punktiga. Põhiversiooni suurenemine tähendab, et uues versioonis on väline rakendusliides muutunud, ning selle vahetamisel tuleb valmis olla oma süsteemi kohaldamiseks. Väikese ja parandusversiooni numbri muutumisel on komponenti võimalik välja vahetada ilma, et peaks enda süsteemi kohaldama. Väikese versiooni erinevus parandusversiooniga võrreldes on, et see sisaldab uut funktsionaalsust, viimane aga ainult parandusi [21].

Kui ettevõte arendab sisest toodet, siis võib versioonitähistuse valik olla liberaalsem, sõltudes uute versioonide väljalaskmise sagedusest. Nädalase arendustsükli puhul võib versiooninumber näiteks koosneda järgmistest osadest: aasta number, nädala number ja räsi versioonihaldusest. Viimane on vajalik, et oleks võimalik lihtsalt siduda ehitatud tehis lähtekoodi versiooniga. Selguse mõttes võib versioonitähisele lisada ka markeeringu tähistamaks, kas tegemist on testimisjärgus või juba stabiilse versiooniga, vastavalt *rc* (*Release Candidate*) ja *stable* (stabiilne). Versiooninumber võiks olla kajastatud ka *AssemblyInfo* failis, mis on justkui DLL-i manifest, mis hoiab endas meta-andmeid nagu versioon ja autori nimi. TeamCitys on selle muutmiseks sisse ehitatud funktsionaalsus, mille leiab *Build Features > AssemblyInfo patcher* alt.

7.3 Veebisaidi erisused

Veebi kliendipoolsete ressursside (skriptid, stiilid) ehitamisel peetakse silmas nende konverteerimist brauserile arusaadavasse keelde (SASS või LESS kompileeritakse CSS-iks või TypeScript kompileerub JavaScriptiks), nende pakkimist (*minify*) ja kombineerimist. Pakkimisel võetakse skriptidest ja stiilidest ära põhimõtteliselt kõik, mida masinal vaja ei ole, erandina võidakse alles jätta litsentsist tulenevalt päis, milles on autori viide. Sisu, mida arvutil inimesega võrreldes ei ole vaja, on tühikud, reavahetused, kommentaarid ja kirjeldavad muutujanimed. Väiksemaid faile üheks failiks kombineerimisel piiratakse brauseri poolt tehtavaid päringute arvu. W3 soovitusel ei tohiks üks klient hoida üleval rohkem kui kahte samaaegset ühendust ühe serveriga [22]. Brauserid, sõltuvalt versioonist, piiravad ühele domeenile tehtavate ühenduste arvu [23].

Üldise tava kohaselt lisatakse failinimele infiks „min“. Ehitamise käigus võib optimeerida ka pildifaile, tihendades neid piisavalt palju, et kvaliteet jääks praktiliselt samaks, aga suurus väheneks.

IIS veebiserverisse on kõige optimaalsem paigaldada ASP.NET MVC põhine veebileht Web Deploy abil. Alternatiiviks on failide käsitsi kopeerimine või läbi IIS administreerimisliidese, mis läheb vastuollu meie eelpool mainitud teesiga vältida inimese sekkumist protsessi. Web Deploy võrdleb lähte- ja sihtfailide ajatempleid ning kopeerib ainult muutunud failid, tehes protsessi üsnagi efektiivseks. Ehitamise käigus tehakse tihendatud zip laiendiga fail, mis hoiab endas kõiki tehiseid ning parameetrite nimesid ja asukohti, mida paigaldamise ajal on vajalik muuta (näiteks andmebaasi ühenduse parameetrid). Veebisaidi puhul tuleks tähele panna, et staatiliste, ehk harva või mitte kunagi muutuvad failid võiksid olla viidatud projektist väljastpoolt. Viimane on kasulik selle tõttu, et hoida pakkimise ajal failide kopeerimise arvelt aega kokku, suuremate projektide puhul võib see säästa isegi minuteid. Ideaalis võiksid tihendatud faili jõuda ainult kombineeritud ja minimeeritud skriptid, stiilid ja optimeeritud pildid.

Näidisinfosüsteemis võttis veebisaidist tihendatud paki loomine aega keskmiselt 50 sekundit. Autor tegi kindlaks harva muutuvad stiilid, skriptid ja pildid ning viis need projektist välja välisele võrgukettale ning viitas need veebisaitides avaliku URL-iga. Tihendatud paki loomine vähenes selle abil 20 sekundit, mis on vajalik ajasääst tegemaks protsessi kiirendamiseks.

8 Konfiguratsioonihaldus

Konfiguratsiooni all peab autor silmas rakenduse seadistamiseks erinevates keskkondades vajalikke parameetreid. Selle haldamiseks on mitu moodust: konfiguratsiooni võib hoida koos lähtekoodiga üldises versioonihalduses ning panna kaasa rakenduse ehitamise ajal või hoida eraldi ning pärida vastavad parameetrid rakenduse paigaldamise ajal. Kuna rakendust testitakse enne toodangusse minekut ka ühes või mitmes toodangulaadses testkeskkonnas, ei ole konfiguratsiooni sidumine rakenduse binaarfailidega hea mõte, sest enne igat paigaldust tuleks sel juhul kas uuesti kompileerida või pakis muudatusi teha. Iga muudatuse tegemine seab omakorda ohtu andmete terviklikkuse, ehk eksisteerib võimalus, et eelnevalt mõnes teises keskkonnas testitud tarkvara võib käituda teisiti. Oluliselt kindlam on hoida konfiguratsioon eraldi. Sedasi on lihtsam varjata toodagu parameetreid isikute eest, kes puutuvad kokku ainult testkeskkondadega. ASP tehnoloogial veebisaidi loomisel on võimalus tekitada *parameters.xml* fail, mis kirjeldab, milliseid parameetreid peab paigaldamise ajal kindlasti ette andma. Web Deploy tarkvara võtab parameetriks tihendatud veebipaki, võti-väärtus stiilis konfiguratsiooni kirjed ja IIS veebiserveri fondi (*pool*).

Rakenduse ehitamisel tekkinud tehiseid võib samuti pidada konfiguratsiooniks. Enamasti PI serverid pakuvad funktsionaalsust erinevate protsesside käigus tekkinud tehiste hoidmiseks. Sõltuvalt ehitatud tarkvara iseloomust tuleks mõelda ka failisüsteemile, sest mõned neist võivad paljude väikeste failide kopeerimisel aeglaseks jääda. Siinkohal päästaks tehiste lisamine tihendatud konteinerisse, näiteks .zip faili. Olenevalt projekti suurusele ja ehitamise tihedusele, tuleb mõelda ka tehiste säilitamise perioodile. Kõvakettaruum ei ole küll tänapäeval enam väga kulukas, kuid ilmselt pole otstarbekas hoida pikema perioodi vältel alles kandidaatversioone.

Analüüsi osas otsustati, et ehitamine tuleb viia eraldi serverisse ning seetõttu on vaja leida sobiv viis transportimaks ehitatud tehised sihtserveritesse.

Autor testis kolme tehiste transportimise viisi:

1. TeamCity haldab ise tehiseid.

2. Tehised pakitakse .zip ja .7z laiendiga ning kopeeritakse võrgukettale.

3. Tehised ehitatakse otse võrgukettale (MSBuild parameeter *OutputPath*).

Tehiseid on kokku üle 2000 faili, kogusuurusega üle 900MB. Mitmete katsete ning erinevate tihendamise astmete proovimise tulemus on järgmine: kõige aeglasemalt sai tehiste transpordiga hakkama TeamCity, kopeerimiseks läks aega üle 4 minuti. Tehiste tihendamine ja kopeerimine võrgukettale võttis aega märksa vähem, 1,5 minutit. Autori suureks üllatuseks ei muutnud tehiste otse võrgukettale ehitamine protsessi oluliselt aeglasemaks, seega autor valis selle variandi kasuks.

9 Testimine

Testimine on tarkvara täitmine / käivitamine kontrollimaks, kas ta vastab ettenähtud nõuetele ning leidmaks vigu [24]. Ühiktestid peaksid olema kiired ning valmima koos arendusega või TDD (*Test-driven development*) puhul isegi enne reaalset funktsionaalsuse kirjutamist. Piisava katvusega hästi kirjutatud ühiktestid annavad osapooltele kindlustunde, et rakenduses tõenäoliselt ei lõhutud viimaste koodimuudatustega midagi ära. Ehk teisisõnu, testimine annab julguse koodis muudatusi teha: nii väikseid veaparandusi kui ka suuremaid koodi korrastamisi (*refactor*). Käitades teste regulaarselt terve koodibaasi vastu, avastades regressioone võimalikult vara, moodustades stabiilseid tehiseid (*artifact*) – nii väldib pidev integreerimine ebaefektiivsust protsessides, mida peetakse agiilse arenduse üheks nurgakiviks [8].

Mida kiiremini on teste võimalik käitada, seda kiirem on tagasiside pärast muudatuse tegemist rakenduses. Vigade parandamine on kõige odavam siis, kui vead kunagi versioonihaldusesse ei jõuagi [7, lk 27]. Esimene kaitseliin on paljude programmeerimiskeelte puhul kompilaator (nii ka C# puhul). Järgmise löögi peaksid kinni võtma juba ühiktestid ning sealt edasi juba aeglasema iseloomuga vastuvõtu- ning staatilised testid, ülevaatamised ja muud kontrollid.

Ühiktestide puhul testitakse programmi üksikuid osasid [24]. Tuntuimaks ühiktesti raamistikuks .NET keskkonnas on NUnit, mis on ka TeamCitys vaikimisi olemas. Samuti on TeamCitys olemas ka veidi teistsuguse süntaksiga testimisraamistiku MSpec (*Machine Specifications*) tugi.

Võib leiduda teste, mis sama lähtekoodi juures käituvad erinevalt. TeamCityl on selliste testide jaoks eraldi funktsioon *Flaky tests,* mis kuvab ebastabiilseid teste. Ühiktestide käitamisele tuleks ette anda ajaline piir, kas siis keskmine aeg ühe testi kohta või absoluutne maksimum. Kui viimane on saavutatud, siis tuleks optimeerida testide käitamist.

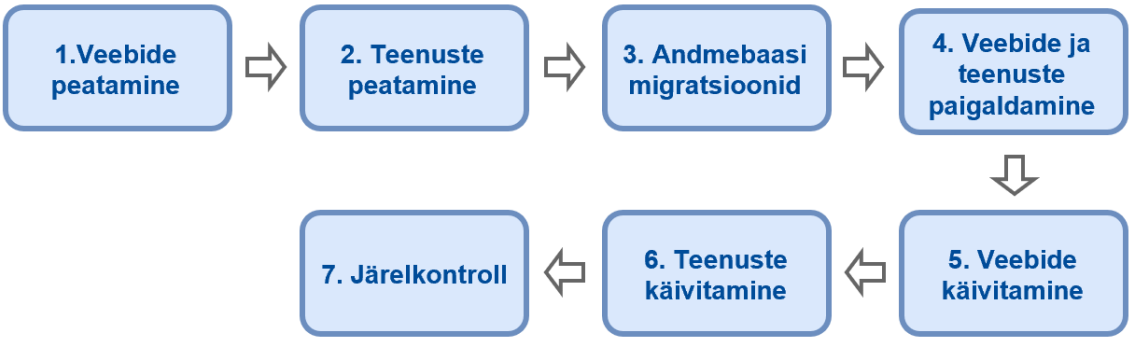
Enne koodimuudatuste avalikustamist ühiskasutatavasse versioonihaldusharusse tuleks käivitada ühiktestid. Viimane nõuab distsipliini, kuid õnneks on olemas tööriistad, näiteks GitLabi mestimissoov (*merge request*), mis võimaldab kasutaja harus käitada ühikteste enne, kui arendaja saab oma muudatused mestida (*merge*) ühiskasutatavasse harusse.

Näidissüsteemis vältab ühiktestide käitamine üle 13 minuti, mis on liiga pikk aeg. Uurides testide käitamise logifaile, järeldub, et 85% testidest kasutavad MSpec raamistikku ning ülejäänud 15% NUnit raamistikku. Dokumentatsioonist järeldub, et MSpec ei toeta vaikimisi testimoodulite paralleelset käitamist. Autor katsetas lisamoodulit *Machine.Specifications.TeamCityParallelRunner,* kuid see tõi kaasa osade testide ebaõnnestumise, kuna nendes oli kasutatud staatilist teeki EasyHook ning paralleelsed pöördumised ei ole toetatud. Ebaõnnestunud testide hulk oli piisavalt suur, et mitte hakata neid ümber kirjutama, seega tuli lahendust otsida mujalt. Seejärel proovis autor käivitada iga MSpeci käsurea protsessi eraldi PowerShelli tööna (sel juhul võiks olla tööl eraldi kontekst ning ehk saab EasyHooki probleemist tänu sellele jagu). Tulemus oli edukas ning testide kogu käitamise aeg kahanes 13 minuti pealt 3,5 minutile (töös käsitletud skriptid on saadaval autori avalikus koodihoidlas aadressil https://github.com/hkarask/Continuous-Integration). Liikudes edasi NUniti juurde selgub, et erinevatest moodulitest käitatakse teste paralleelselt juba vaikimisi. Lisaks on NUnitil võimalus märkida paralleelseid teste ka mooduli siseselt, mis aga suhteliselt väikese NUniti osakaalu tõttu kogu testidest jäi seekord tegemata [25].

Vastuvõtutestide käitamine võttis näidissüsteemis aega 53 minutit. Vaadates testide koodi selgus, et need on arendatud kõik ühe mooduli piires, mida käitab NUnit. Üsna lihtsa vaevaga sai mooduli poolitada ning nüüd on võimalik mõlemad osad käitada paralleelselt. Vastuvõtutestide läbimine võttis pärast optimeerimist aega 34 minutit, mis teeb kiiruse kasvuks 35%.

10 Paigaldamine

Paigaldamise protsessis võetakse eelnevalt ehitatud ning testitud tehised ja kopeeritakse need vastavatesse sihtpunktidesse. Olenevalt projekti keerukusest võib tegu olla lihtsalt paari faili kopeerimisega. Keerulisema puhul Windowsi teenuste ja IIS saitide peatamisega, installeerimisega, andmebaasi skeemi uuendamisega ja muu säärasega. Allpool on esitatud keskmise SaaS tüüpi infosüsteemi paigaldusprotsess (vaata Joonis 15).



Joonis 15. SaaS tüüpi infosüsteemi paigaldamisprotsessi sammud.

Rakendust on enne toodangusse paigaldamist soovitatav testida toodangulaadses testkeskkonnas. Toodangulaadses sel tingimusel, et testimine peegeldaks võimalikult palju reaalset süsteemikasutamist. Sama kehtib ka paigaldusprotsessi korral. Selleks, et suurendada tõenäosust edukaks toodangu paigalduseks, tuleb seda protsessi enne läbi testida ning vead siluda.

Üldiselt iga sekund, kui toode ei ole lõppkasutaja jaoks saadaval, on ettevõttele kahjumlik. Eriti siis, kui ettevõte üritab leida endale uusi kasutajaid, kes pärast esimest negatiivset kogemust ei pruugi enam toote juurde tagasi tulla. Tuleb leida viis kuidas tõsta süsteemi käideldavust, tõstes komponentide liiasust ja/või kasutades rohkem ressursse, et teha paigaldamine kiiremaks. Liiasusega käideldavuse tõstmine on üsna lihtsasti saavutatav veebisaitide puhul, kus sama versiooniga veebisaidid on pöördproksi taga. Viimane reguleerib olenevalt seadistusest liiklust saitide vahel. Veebide uuendamisel saab need ükshaaval pöördproksi küljest lahti ühendada, rakenduse uuele versioonile viia ning seejärel tagasi ühendada. Mingi aja vältel näevad küll erinevad kasutajad tarkvara eriversioone, kuid see on üldiselt vastuvõetav. Sarnast lähenemist saab kasutada testimaks

uut funktsionaalsust väikese segmendi peal, enne kui see tehakse kõigile kättesaadavaks (*canary release*). Ettevõtted võivad seda kasutada näiteks olukorras, kus uus funktsionaalsus on veel toores ning pakkuda tehnoloogiahuvilistele klientidele võimalust näha uusi lahendusi. Ettevõte saab väärtuslikku tagasisidet, silub toodet ning laseb seejärel toote välja üldsusele.

Paigaldamise protsessis on tihti kaasatud hulk erinevat tüüpi tarkvara ning suhtlus käib mitme erineva serveri vahel. Tõenäosus, et üks neist komponentidest võib põhjustada vea, on päris suur. Failisüsteemis võib esineda vigu, olukord et protsessid võivad mäluprobleemide tõttu kinni kiiluda ei ole harv nähtus. Selliste probleemide vastu on raske proaktiivseid meetmeid rakendada. Reaktiivsetega on olukord lihtsam – päästab protsesside erinevates sammudes logimine. Logifaile tuleks hoida alles mõistliku perioodi vältel, need võivad tulevikus osutuda vajalikuks auditeerimisel. Logidest peaks selguma isik, kes käivitas protsessi ning täpsed kellaajad.

Paigalduse protsessi tehakse erinevates keskkondades ning sellega kokku puutuvatel isikutel peaks olema neis eraldi õigused. Testkeskkond ei ole üldiselt kuigi ärikriitiline ning seal võib vähemalt käivitusõiguse anda enamikele, kel seda võiks vaja olla. Muutmisõigus võiks jääda väiksema grupi kätte, sest üldiselt ei tunne kõik protsessi samal tasemel ning tõenäosus vea tegemiseks süsteemis on väiksem. Toodangu keskkonnas peaks õigustega olema tagasihoidlikum, muutmisõigus peaks olema koondunud mõne üksiku kätte, sama kehtib ka paigaldamise õiguse kohta. Kui süsteem on selgelt jaotunud erinevate meeskondade vahel, võib mõelda ka õiguste sarnase jaotuse peale.

Uue redaktsiooni edukas testimine testkeskkondades ei garanteeri õnnestumist toodangukeskkonnas. Viimases on siiski sageli mingid erinevused, tihti tingitud suurema koormuse tõttu. Vea ilmnemisel on üldiselt kaks valikut: teha kiirpaik (*hotfix*) või minna tagasi eelmisele versioonile (*rollback*). Üldiselt eelistatakse esimest varianti, kuna erisused testkeskkondade ja toodangu vahel on väikesed ning seega võib järeldada, et ka parandus ei saa kuigi suur olla. Eelmisele versioonile tagasi minek on üldiselt ka psühholoogiliselt raskem, kuna kasutajad võivad olla juba uut tarkvara näinud. Sageli ei testita tagasipööramist, mis muudab selle ka üsna riskantseks ettevõtmiseks.

Paigaldust on võimalik teha käsitsi, kuid see on taunitav seetõttu, et protsessi on kaasatud inimfaktor. Inimesed eksivad, ning ei pruugi protsessi korrata alati samasuguselt.

Paigaldamiseks leidub ka erinevaid teenuseid, näiteks Octopus Deploy, kuid see tähendab sõltumist kolmandast poolest. PowerShell on ennast tõestanud võimsa skriptimiskeelena ning paigaldamise ülesehitamine selle peale annab läbipaistvuse ning võimaluse seda oma vajaduste järgi kohandada.

PowerShell toetab skriptide käitamist kaugteel, kasutades selleks WS haldus protokolli (*Web Services Management*) ja WinRM teenust (*Windows Remote Management*). WinRM on vaikimisi paigaldatud alates Windows versioonidest Server 2008 ja 7 [26].

10.1 IIS Veebisait

Veebisaidi paigalduse puhul on reeglina tarvis järgmisi samme:

1. Veebisaidi esmane seadistus.
2. Vastava fondi (*pool*) peatamine.
3. Paki paigaldamine *Web Deploy* abil.
4. Vastava fondi (*pool*) käivitamine.

Sammu nr 1 tuleb teha ainult esmakordsel seadistamisel. Kõik sammud on võimalik teostada läbi graafilise liidese, kuid tuleks kohe alguses alustada selle tegemist läbi skripti, et välistada inimfaktorit. Lihtsaim veebifondide (*app pool*) peatamine on teostatav käskluste abil:

```
Invoke-Command { Stop-WebAppPool VeebA } –ComputerName VS1
Invoke-Command { Stop-WebAppPool VeebB } –ComputerName VS1
Invoke-Command { Stop-WebAppPool API } –ComputerName VS1
Invoke-Command { Stop-WebAppPool VeebA } –ComputerName VS2
Invoke-Command { Stop-WebAppPool VeebB } –ComputerName VS2
Invoke-Command { Stop-WebAppPool API } –ComputerName VS2
```

Antud käsklused peatavad IIS veebisaidid VeebA ja VeebB ning API veebiserverites VS1 ja VS2. Vaadates PowerShelli dokumentatsiooni lähemalt, selgub et nii veebifondi kui ka masina nime parameetrid toetavad loendi (*list*) tüüpi muutujat. Seega saab käskluse lühendada ühe reani:

```
Invoke-Command { Stop-WebAppPool VeebA,VeebB,API } –ComputerName VS1,VS2
```

Tehes mõned katsetused, selgub et mõnedel juhtudel paigaldamine ebaõnnestub kuna veebisait ei ole peatatud ning osa faile on veel kasutuses. Käsu dokumentatsiooni uurimisel seda küll ei selgu, aga internetis on viiteid, et peatamiskäsk (analoogselt ka käivitamis) ei oota ära seisundi muutust. See ei järgi sama lähenemist Windows teenuse puhul. Ohverdades natuke keerukust ja paralleelsust, saame järgmise skripti:

```
foreach ($fond in ('VeebA','VeebB','API')) {
    Invoke-Command -ComputerName VS1,VS2 -ScriptBlock {
        if ((Get-WebAppPoolState $Using:fond).Value -eq 'started') {
            Stop-WebAppPool $Using:fond
            do
            {
                Start-Sleep -Seconds 1
            }
            until ((Get-WebAppPoolState $Using:fond).Value -eq 'stopped')
        }
        else {
            Write-Host "Fond $($Using:fond) on peatatud"
        }
    }
}
```

Lisatud on kontroll, mis vaatab ühe sekundiste intervallide tagant, kas veebisait on peatatud, vastasel korral oodatakse ning proovitakse uuesti. Kui mingil põhjusel esineb, et fondi ei olegi mõistliku aja jooksul võimalik käivitada, tuleks skripti kohandada proovimiste arvu ülempiiriga ning selle täitumisel viga logida ning protsess lõpetada. Tehes uusi teste veendub autor, et eelpool mainitud viga paigaldamisel enam ei teki. Teoorias võib õnnestuda paigaldus ka ilma fondi peatamiseta, kuid praktikas toob see kaasa varikopeerimise (*shadow copy*) probleeme.

Pärast veebifondi peatamist saab alustada tehiste uuendamisega. Tihendatud veebipakkide uuendamine läbi käsurea toimub mooduli Web Deploy abil (tuntud ka kui MSDeploy), mida IIS installatsiooniga vaikimisi kaasas ei ole. Paki paigaldamine PowerShelliga käib läbi *Restore-WDPackage* käsu:

```
foreach ($fond in ('VeebA','VeebB','API')) {
    Invoke-Command -ComputerName VS1,VS2 -ScriptBlock {
        Restore-WDPackage `
            -Package "//paigaldamine/pakid/veeb/1.0.0.0/$fond.zip" `
            -Parameters @{ 'Võti'='Väärtus' }
    }
}
```

Parameetrina saab käsklus ette ehitamise käigus loodud tihendatud paki ning võti-väärtus paaridena parameetrid (mis asendatakse failis Web.config). Käsklus tagastab kokkuvõte sooritatud tegevustest (info aja ning muudatuste hulga kohta). Oluline on, et ehitamise sammus oleks tihendatud pakk üles seatud nii, et kaasas oleks manifest kõikidest nõutud parameetritest, mida veebil on tööks on vaja, vastasel juhul ei pruugi rakendus sihipäraselt töötada.

Sammus 4 käivitame vast uuendatud veebi analoogselt peatamisega:

```
foreach ($fond in ('VeebA','VeebB','API')) {
    Invoke-Command -ComputerName VS1,VS2 -ScriptBlock {
        if ((Get-WebAppPoolState $Using:fond).Value -eq 'stopped') {
            Start-WebAppPool $Using:fond
            do
            {
                Start-Sleep -Seconds 1
            }
            until ((Get-WebAppPoolState $Using:fond).Value -eq 'started')
        }
        else {
            Write-Host "Fond $($Using:fond) on käivitatud"
        }
    }
}
```

Näidissüsteemis (vaata Joonis 3) on määratud, et veebisaidid on dubleeritud ning asuvad koormusjaoturi taga. Seega, kui meil on võimalik käitada korraga vana ja uut versiooni veebirakendusest (näiteks andmebaasi struktuuri muutused võimaldavad seda), saab uue redaktsiooni lansseerida ilma, et rakendus oleks kordagi lõppkasutaja jaoks kättesaamatu. Olenevalt koormusjaoturi seadistusest (ei kuulu antud töö skoopi), peame enne veebisaidi uuendamist ühes serveris (näiteks VS1) selle koormusjaoturi küljest lahti ühendama, et sinna ei tuleks uusi päringuid. Pärast veebi uuendamist masinas VS1 saame koormusjaoturile öelda, et veeb on jälle valmis kasutamiseks ning tuleb lahti ühendada veeb teisest masinast, ehk VS2-st. Sellisel viisil uuendati mõlemad veebisaidid ilma, et

kasutaja oleks tajunud seisakuid. Teine variant oleks koormusjaotur seadistada nii, et kui üks veebisait ei vasta kindla aja jooksul, suunatakse päring teise masina samasse veebi. Sellise lähenemise negatiivseks küljeks on, et esimesed päringud võtavad kaua aega ning on lõppkasutajale ebamugavad.

Pärast paigaldust tuleks teha lihtne suitsutestimine, näiteks GET päring avalehele kontrollimaks, et veebisait suudab vastata päringule.

10.2 Windows teenus

Windows teenuse paigaldamine on analoogne veebipaigalduse protsessile:

1. Teenuse peatamine.
2. Teenuse failide puhastamine.
3. Teenuse failide kopeerimine.
4. Konfiguratsiooni kohaldamine (peab toimuma siin sammus).
5. Teenuse esmane seadistus (installeerimine, käivitustüübi määramine).
6. Teenuse käivitamine.

Ka siin saab mõningaid samme teha edukalt läbi graafilise liidese, kuid see ei ole jätkusuutlik ning tuleks pöörduda kohe skriptimise juurde. Teenuse paigaldamine saab alguse selle peatamisest, PowerShelli käsk selleks on järgmine:

```
Stop-Service TeenuseNimi
```

Erinevalt veebifondist, oodatakse siin teenuse peatamist enne, kui skripti täitmist jätkatakse, ning lisakontrolle juurde ehitada tarvis ei ole. Näidissüsteemis on märgitud, et mõlemasse rakendusserverisse on paigaldatud 20 teenust Nende peatamine ükshaaval on väga ajakulukas. Oletame, et ühe teenuse peatamiseks kulub 30 sekundit, siis kulub ühes serveris kõikide teenuste peatamiseks 600 sekundit ehk 10 minutit, mis on ilmselgelt liiga pikk aeg. Siinkohal on mõistlik kasutada taustal käivaid PowerShelli töid (*job*). Järgmises skriptinäites on kaks loendit teenustest rakendusserveris RS1 ja RS2 ning nende paigaldamine tööde abil.

```
# Teenused rakendusserveris RS1
$RS1Teenused = @('RS1T1', 'RS1T2', 'RS1T3'..'RS1T10')
# Teenused rakendusserveris RS2
$RS2Teenused = @('RS2T1', 'RS2T2', 'RS2T3'..'RS2T10')

foreach ($teenus in $RS1Teenused) {
    Invoke-Command -ScriptBlock { Stop-Service $using:teenus } `
    -ComputerName RS1 -AsJob
}
foreach ($teenus in $RS2Teenused) {
    Invoke-Command -ScriptBlock { Stop-Service $using:teenus } `
    -ComputerName RS2 -AsJob
}
Get-Job | Wait-Job
```

Iga käivitamise käsust tehakse PowerShelli töö, mille käitlemine toimub taustal. Skripti lõpus oodatakse kõikide tööde lõpetamist. Lõpetanud tööde saabumisel peab kontrollima tööde staatust. Kui see on midagi muud peale staatuse „õnnestunud“ (*Completed*), siis tuleks vaadata tulemuse sisu. Samuti tuleks puhastada tööde loend *Remove-Job* käsu abil.

Autor testis eelpool kirjeldatud teenuste paigaldamise skripti. Testimise käigus tekkis kümnest katsest kahel WinRMi viga, mille juurpõhjust ei suutnud autor välja selgitada. Autor kahtlustab, et 40 töö korraga tekitamine võib WinRMi üle koormata. Mõistlik oleks samaaegselt käitavaid töid hoida mingis etteantud piirväärtuses. Selline arv võiks olla seoses masina võimsusega, näiteks loogiliste protsessorite arv korda kaks:

```
$paralleelseidToid = 2 * (Get-WmiObject -class Win32_processor
       | measure NumberOfLogicalProcessors -Sum | select -ExpandProperty Sum)
```

Täiustades eelmist skriptinäidet, saame:

```
$parallelseidToid = 2 * (Get-WmiObject -class Win32_processor |
    Measure-Object NumberOfLogicalProcessors -Sum |
    Select-Object -ExpandProperty Sum)

function lisaToo ($sb, $machine) {
    $toidKaimas = Get-Job -State Running
    if ($toidKaimas.Count -ge $nrOfConcurrentJobs) {
        $toidKaimas | Wait-Job -Any
    }
    Invoke-Command $sb -cn $machine -AsJob
}

$RS1Teenused = @('RS1T1', 'RS1T2', 'RS1T3'..'RS1T10')
$RS2Teenused = @('RS2T1', 'RS2T2', 'RS2T3'..'RS2T10')

foreach ($teenus in $RS1Teenused) {
    lisaToo { Stop-Service $using:teenus } RS1
}
foreach ($teenus in $RS2Teenused) {
    lisaToo { Stop-Service $using:teenus } RS1
}
Get-Job | Wait-Job
```

Enne uue töö alustamist toimub eelkontroll, mis jälgib, et paralleelselt töös olevaid käske ei oleks rohkem kui etteantud piirmäär. Kümne testimiskorra jooksul ei suutnud autor enam eelpool mainitud WinRM viga saada. Käies jadamisi läbi teenuste nimekirja ning piirates paralleelseid töid, tekib aga uus probleem. Nimelt ei ole tööde samaaegne teostamine hajutatud kahe serveri vahel. Oletame et, paralleelsete tööde arv on 4, siis võtab skript esimesest teenuste loendist (RS1Teenused) neli ning ootab, millal üks neist lõpetab, ning võtab siis järgmise. Skripti algul käib kogu töö masinas RS1 ning lõpus ainult masinas RS2. Protsess oleks oluliselt efektiivsem, kui saaksime töö jaotada igal ajahetkel serverite vahel võrdselt. Eeldusel, et teenuste jadad on sama pikad, saame lisada eelmisele skriptinäitele lihtsa koormuse jaotamise:

```
$parallelseidToid = 2 * (Get-WmiObject -class Win32_processor |
    Measure-Object NumberOfLogicalProcessors -Sum |
    Select-Object -ExpandProperty Sum)

function lisaToo ($sb, $machine) {
    $toidKaimas = Get-Job -State Running
    if ($toidKaimas.Count -ge $nrOfConcurrentJobs) {
        $toidKaimas | Wait-Job -Any
    }
    Invoke-Command $sb -cn $machine -AsJob
}

$RS1Teenused = @('RS1T1', 'RS1T2', 'RS1T3'..'RS1T10')
$RS2Teenused = @('RS2T1', 'RS2T2', 'RS2T3'..'RS2T10')

foreach ($i in 0..($RS1Teenused.Length - 1)) {
    $teenus = $RS1Teenused[$i]
    lisaToo { Stop-Service $using:teenus } RS1
    $teenus = $R21Teenused[$i]
    lisaToo { Stop-Service $using:teenus } RS2
}
Get-Job | Wait-Job
```

Pärast muudatusi jaotub koormus ühtlasemalt kahe rakendusserveri vahel ning paigaldusprotsess on kiirem.

Pärast Windows teenuste peatamist saab alustada tehiste puhastamisega, pärast mida saab uued tehised kopeerida. See samm on üldiselt üsna kiire ning ei vaja täpsemat optimeerimist, kuid soovi korral saab siin võrrelda enne kustutamist failide loendit, ning uuendada ainult need mille ajatempel on erinev.

Konfiguratsiooni kohaldamisel loetakse sisse XML põhine App.config fail (või ka mingi muu, nt JSON) ning siis saab juba mugavalt teostada muutmisoperatsiooni PowerShelli XML dokumendi objekti põhjal. Siin sammus muudetakse tavaliselt andmebaasi ühendused ja muud teenuste parameetrid.

Viiendas sammus installeeritakse teenus (kui see on uus) ning seatakse sobiv käivitamisrežiim. See võib olla manuaalne, automaatne või viitega automaatne. Manuaalse korral peab masina käivitamisel ise juhtima protsessi, mis käivitaks teenuse. Automaatse korral käivitub teenus kohe pärast masina käivitumist. Viitega teenuse korral oodatakse enne kõigi automaatsete teenuste käivitumist. Viimane on kasulik juhul, kui teenus sõltub teisest teenusest, näiteks samas masinas olevast andmebaasist. Automaatse käivitamise puhul on probleemiks, et teenused käivituvad ükshaaval ning eelpool toodud

näite puhul võtab see suurema teenusekomplekti korral liiga kaua aega. Selle põhjal võib väita, et efektiivsem oleks käivitamisskript haakida külge masina käivitamissündmusele. Viimases sammus käivitame teenused analoogselt eespoolt kirjeldatud peatamisega.

Paar minutit pärast paigaldust tuleks testida, et süsteem on soovitud olekus, vaadates teenuste parameetreid (kas teenus veel käib, palju mälu kasutab jne).

10.3 Andmebaasi migratsioonid

Rakendus salvestab tavaliselt oma seisu mõnda andmehoidlasse (*data storage*), olgu see siis failisüsteem või andmebaas. Tihti on selleks andmehoidlaks mõni relatsioonile andmebaas, näiteks Microsoft SQL server või MySQL, mis on ennast juba lahingus tõestanud. Kui NoSQL andmebaas on andmete struktuuri osas üsna leplik, siis relatsioonilise andmebaasi korral tuleb valmistada baas ette DDL (*Data Definition Language*) skriptide abil. Skriptide käsitsi paigaldamine on üsna tülikas ning võib põhjustada inimlikke vigu (käitatakse vales järjekorras või unustatakse mõni skript ära). On selge, et kogu protsess tuleb automatiseerida ning hoida koos ülejäänud lähtekoodiga versioonihalduses. Migratsioonid võib kirjutada nii SQL keeles endas või kasutada mõnda teeki mis pakub enda liidestust (näiteks FluentMigrator või Entity Framework Code First Migrations). Olenemata valikust, tuleb automatiseerida protsess, mis rakendab migratsioonid. Trigeri võib näiteks siduda arenduskeskkonnas ehitamise protsessiga (näiteks MSBuild task) või siis versioonihaldustarkvara haagiga (*hook*), mis käitatakse uue lähtekoodi redaktsiooni küsimisel kesksest hoidlast.

Paigaldamise töövoos peab üldiselt andmebaasi migratsioonid läbi viima siis, kui rakendus on peatatud. Põhjus selles, et relatsioonilise andmebaasi puhul eeldab infosüsteem kindlalt struktuuri ning selle muutmisel võib rakenduses tekkida vigu. Enne paigaldamist tuleks analüüsida ka migratsioonide kestvust, väga mahuka tabeli muutmisel võib protsess väldata väga kaua. Samuti tuleks teha ka pöördmigratsioon olukorraks, kus on vaja minna tagasi eelmisele rakenduse versioonile (*rollback*). Kasutades teeki FluentMigrator, sunnib see kasutajat looma meetodid migreerimaks järgmisele versioonile (*Up*) ning tagasi eelmisele (*Down*). Siinkohal tuleb jällegi mängu arendajate distsipliin, sest eelpool mainitud funktsioonid on võimalik ka tühjaks jätta.

10.4 Juhtprotsess

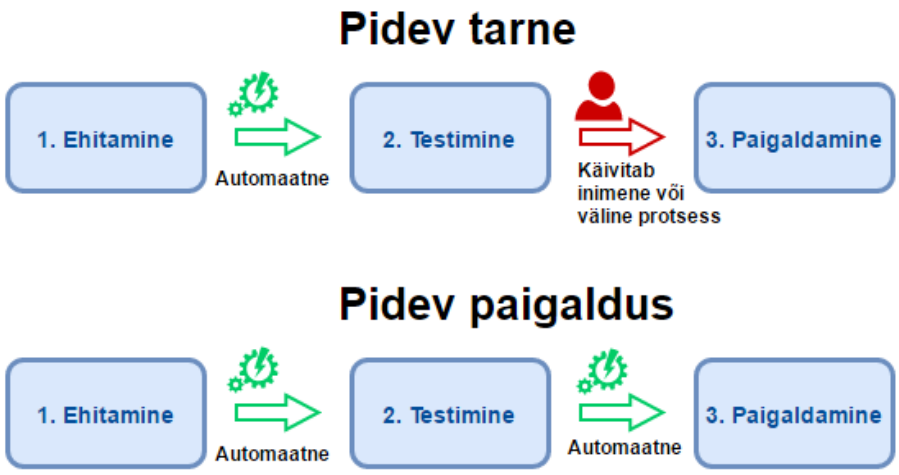
Käesolevas peatükis on selgitud paigaldamise erinevaid alamprotsesse ning nüüd tuleks need juhtprotsessi abil kokku siduda üheks tervikuks. Peatüki alguses oleval joonisel (vaata Joonis 15) on graafiliselt esitatud lineaarne paigaldamise protsess. Seda saaks jällegi teostada PowerShelli abil või kirjeldada TeamCity ehitamissammude abil. Viimane pakub veaolukordades paremat ülevaatlikust ning erinevaid samme on lihtne sisse ja välja lülitada.

Tehes kümme paigaldust, tuli keskmise paigalduse ajaks 6 minutit, mis on ligi 4 korda parem algsest tulemusest. Lisaks protsessi kiiruse paranemisele, ei tekkinud paigaldamise käigus süsteemis ühtegi viga. Tänu erinevatele optimeerimistele, suudeti protsess teha oluliselt kiiremaks, lihtsamaks ning ka veakindlaks.

11 Pidev tarne ja paigaldamine

Valmis arendatud funktsionaalsus ei teeni ettevõttele raha enne kui see on paigaldatud ja lõppkasutajale kättesaadav. Seega võib väita, et tsükli pikkus funktsionaalsuse valmimisest kuni paigaldamiseni peab olema võimalikult lühike, et maksimeerida kasumlikkust. Pidev tarne tähendab valmisolekut igal hetkel paigalduse läbiviimiseks, võttes tehised eelnevalt õnnestunud pideva integratsiooni läbinud protsessist. Paigaldamise protsessi võib käivitada kindlaks määratud ajaga triger või inimene. Ehkki protsessi võib käivitada inimene, tuleb siiski meeles pidada, et käivitamine ei tähenda protsessi läbi viimist, ehk protsess paigaldamise siseselt peab olema automatiseeritud.

Kui pidev tarne on valmisolek igal hetkel paigaldada, siis pidev paigaldamine tähendab, et see tehakse automaatselt kui koodimuudatus on läbinud kõik automaatsed kontrollid. Järgnev joonis illustreerib hästi pideva tarne ja paigaldamise protsessi erinevust (vaata Joonis 16).



Joonis 16. Pidev tarne ja paigaldus.

Pidev paigaldamine on küpsusaste, mis eeldab kõikide manuaalsete sammude eemaldamist ning on keerulisema süsteemi puhul raske saavutada. See eeldab, et automaattestide loomine on arenduse lahutamatu osa, millele pannakse palju rõhku. Sarnaselt automaatsele paigaldamisele võib teha ka automaatset eelmisele versioonile tagasipööramist, eeldusel et tagasipööre eelmisele versioonile on põhjalikult testitud ning ei põhjusta segadust lõppkasutajate seas.

Antud näidissüsteem sõltub tugevalt manuaalsest testimisest ning pideva paigalduseni jõudmiseks tuleb manuaalsete vastuvõtutestide automatiseerimiseks palju tööd teha.

12 Töö tulemused

Käesoleva töö tulemusel suudeti kõik analüüsi käigus püstitatud eesmärgid saavutada. Ebamäärasest algprotsessist sai selgelt defineeritud lineaarne töövoog, mis juhib paremini arendusetappides avalduda võivaid riske. Autor tuvastas süsteemis olevad kitsaskohad ning suutis need rahuldava tasemeni optimeerida. Selle tulemusena paranes süsteemi ehitamise aeg 77%, ühiktestide käitamise aeg 73% ja vastuvõtutestide aeg ligi poole võrra. Tänu ehitamise lahutamisele paigaldamisest ning protsesside viimisele paralleelseks, suutis autor viia paigaldamise aja kuuele minutile, mis on neljakordne langus võrreldes algse 24 minutiga. Paigaldamist saab nüüd teha käivitades ühe juhtprotsessi, mis koordineerib paigaldust kõikidel serveritel. Tulemuseks on täpselt juhitud protsess, mis tagab süsteemi eri komponentide õigeaegse peatamise ja käivitamise, et vältida vigu rakenduse töös. Selle tagajärjel ei teki paigalduse tõttu enam süsteemis vigu.

Tulevikus saab ehitamisprotsessi kiiremaks teha lisades ehitamismasinasse rohkem protsessori ressurssi. Ehitamise osas mainitud MSBuildi seaded võimaldavad vajaminevat lisaressurssi automaatselt kasutusele võtta. Sama kehtib ka ühiktestide käitamisega – lisa tuumade arvu kasutusele võtmisega on võimalik üsna lihtsalt testide käitamise aega vähendada. Paigaldamise protsessis kulus kõige rohkem aega teenuse käivitamisele ning ka see oli otseselt protsessori tuumade arvuga seotud. Teenused on autori arvates otstarbekas riskide maandamiseks hajutada rohkemate masinate vahel, kui ühe masinaga peaks midagi juhtuma, on mõjutatud väiksem arv teenuseid.

13 Kokkuvõte

Agiilsete arendusmetoodikate peavoolu jõudmisega on rakenduste tarnimise tsükkel tunduvalt lühenenud, mõnel juhul tarnitakse isegi mitu korda päevas. Kiire tarnimisvõimalus võib ettevõttele anda turueelise ja uutele tingimustele kiire kohanemisvõime. Juhtimaks rakenduse tarnimisel tekkivaid riske, on vaja efektiivset protsessi.

Magistritöö sai alguse autori isiklikust huvist IT valdkonnas töötades ja ideest edendada tarkvaraarenduse protsessi tervikuna. Märgates, et tema igapäevatöös üks tehniliselt ajakulukaimaid protsesse on just tarkvara integratsioon ja tarnimine, siis otsustas autor probleemi süvitsi uurida.

Käesoleva magistritöö eesmärgiks on tarkvaraarenduse protsessi tõhustamine pideva integratsiooni ja tarne kaudu. Kuna Windowsil on teiste operatsioonisüsteemidega võrreldes märkimisväärne turuosa, otsustas autor uurimist kitsendada just sellel platvormil. Uue protsessi loomiseks võttis autor aluseks Windowsi põhise SaaS (*Software as a Service*) tüüpi rakenduse, mis on loodud .NET raamistikul. Autor analüüsis näidissüsteemi ning esitas ekspertteadmiste põhjal hüpoteesid parenduste tegemiseks. Seejärel analüüsis autor protsessi iga sammu teoreetilist poolt levinud praktikate näitel ning leidis empiirilisel teel parima lahenduse protsessi täiustamiseks.

Magistritöö tulemusteks on efektiivsem protsess, selle realisatsioon ja juhised. Kõik püstitatud eesmärgid saavutati ning probleemile leiti lahendus. Protsessi igas sammus tehtud optimeerimised tõid kaasa tuntava kiiruse kasvu ja veakindluse. Antud töö sobib kasutamiseks ka juhistena igas suuruses ja valdkonnas ettevõtetele. Juhistest on abi nii olemasolevate kui ka uute infosüsteemide puhul, mis soovivad kasu saada efektiivsest pideva integreerimise ja paigalduse protsessist.

Kasutatud kirjandus

[1] C. Rossi, „Ship early and ship twice as often,“ 03.08.2012. [Võrgumaterjal]. https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920/. [Kasutatud 09.04.2017].

[2] W3Techs, „Usage of operating systems for websites,“ [Võrgumaterjal]. https://w3techs.com/technologies/overview/operating_system/all. [Kasutatud 09.04.2017].

[3] agilemanifesto.org, „Principles behind the Agile Manifesto,“ [Võrgumaterjal]. http://agilemanifesto.org/iso/et/principles.html. [Kasutatud 21.03.2017].

[4] Mozilla Corporations, „Firefox Releases,“ [Võrgumaterjal]. https://www.mozilla.org/en-US/firefox/releases. [Kasutatud 27.04.2017].

[5] Google Chrome team, „Chrome Releases,“ [Võrgumaterjal]. https://chromereleases.googleblog.com. [Kasutatud 27.04.2017].

[6] M. Fowler, „AgileVersusLean,“ 26.06.2008. [Võrgumaterjal]. https://martinfowler.com/bliki/AgileVersusLean.html. [Kasutatud 21.03.2017].

[7] J. Humble ja D. Farley, Continuous Delivery, Boston: Addison Wesley, 2010.

[8] K. Sandoval, V. V. D. Mersch ja C. Wood, API-Driven DevOps: Strategies for Continuous Deployment, Leanpub, 2016.

[9] R. Wilsenach, „DevOpsCulture,“ 09.07.2015. [Võrgumaterjal]. https://martinfowler.com/bliki/DevOpsCulture.html. [Kasutatud 01.05.2017].

[10] M. Fowler, „SnowflakeServer,“ 10.07.2012. [Võrgumaterjal]. https://martinfowler.com/bliki/SnowflakeServer.html. [Kasutatud 20.04.2017].

[11] Y. Izrailevsky ja A. Tseitlin, „The Netflix Simian Army,“ 19.07.2011. [Võrgumaterjal]. http://techblog.netflix.com/2011/07/netflix-simian-army.html. [Kasutatud 23.04.2017].

[12] M. Fowler, „MicroservicePremium,“ 13.04.2015. [Võrgumaterjal]. https://martinfowler.com/bliki/MicroservicePremium.html. [Kasutatud 01.04.2017].

[13] K. Sutton, „Monoliths, Microservices, and MVCx2,“ 26.03.2015. [Võrgumaterjal]. http://kellysutton.com/2015/05/26/monoliths-microservices-and-mvcx2.html. [Kasutatud 01.05.2017].

[14] JetBrains s.r.o., „Configuration,“ [Võrgumaterjal]. https://www.jetbrains.com/teamcity/features/configuration.html. [Kasutatud 01.04.2017].

[15] Jetbrains s.r.o., „User Management,“ [Võrgumaterjal]. https://www.jetbrains.com/teamcity/features/user_management.html. [Kasutatud 01.04.2017].

[16] P. Kelcey, „Repo-driven build triggers, the quiet period, and you,“ [Võrgumaterjal]. https://www.atlassian.com/continuous-delivery/understanding-the-quiet-period. [Kasutatud 27.04.2017].

[17] Microsoft, „Incremental Builds,“ [Võrgumaterjal]. https://msdn.microsoft.com/en-us/library/ee264087.aspx. [Kasutatud 24.04.2017].

[18] Microsoft, „NuGet 1.0 and 1.1 Release Notes,“ [Võrgumaterjal]. https://docs.microsoft.com/en-us/nuget/release-notes/nuget-1.1. [Kasutatud 25.04.2017].

[19] DevelopersSite.org, „Preventing referenced assembly PDB and XML files copied to output,“ [Võrgumaterjal]. http://www.developersite.org/101-43375-xml. [Kasutatud 25.04.2017].

[20] Microsoft, „Building Multiple Projects in Parallel with MSBuild,“ [Võrgumaterjal]. https://msdn.microsoft.com/en-us/library/bb651793.aspx. [Kasutatud 25.04.2017].

[21] T. Preston-Werner, „Semantic Versioning 2.0.0,“ [Võrgumaterjal]. http://semver.org/. [Kasutatud 25.04.2017].

[22] W3.org, „Hypertext Transfer Protocol -- HTTP/1.1,“ [Võrgumaterjal]. https://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html. [Kasutatud 24.04.2017].

[23] Browserscope, „Network,“ [Võrgumaterjal]. http://www.browserscope.org/?category=network. [Kasutatud 01.04.2017].

[24] British Computer Society Specialist Interest Group in Software Testing, „Living Glossary,“ [Võrgumaterjal]. http://www.testingstandards.co.uk/living_glossary.htm#T. [Kasutatud 26.04.2017].

[25] C. Poole, „Parallel Test Execution,“ 03.04.2017. [Võrgumaterjal]. https://github.com/nunit/docs/wiki/Parallel-Test-Execution. [Kasutatud 03.05.2017].

[26] TechNet, „Enable and Use Remote Commands in Windows PowerShell,“ [Võrgumaterjal]. https://technet.microsoft.com/en-us/library/ff700227.aspx. [Kasutatud 30.04.2017].