TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Joosep Põllumäe 130441IABMM

# DEVOPS PÕHIMÕTETE JUURUTAMINE AGIILSES TARKVARAARENDUSPROTSESSIS ELISA EESTI AS NÄITEL

Magistritöö

Juhendaja: Villu Teearu
Magister

Tallinn 2019

# 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: Joosep Põllumäe

12.05.2019

# Annotatsioon

Magistritöö keskendub Elisa Eesti AS agiilses arendusprotsessis DevOps arendusmetoodika ja põhimõtete realiseerimiseks vajalike infotehnoloogiliste lahenduste leidmisele ning nende mõju analüüsimisega tarkvaraarhitektuurile ning tööprotsessidele erinevates tarkvaraarenduse protsessi etappides. Kvalitatiivse uuringu olulisemad järeldused on, et tarkvara arendamisel tuleb lähtuda mikroteenusarhitektuurist ning rakenduste automaatseks tarnimiseks erinevatesse keskkondadesse on vajalik kasutusele võtta Docker konteinerid. Lisaks leitakse, et tarkvara automaatseks tarnimiseks on vajalik „taristu kui teenus“ tüüpi pilvelahendus ning Docker konteinerite orkestreerimise platvorm. Nimetatud lahenduste kasutusele võtmine võimaldab senised manuaalsed protsessid automatiseerida ning see läbi väheneb arenduste *time-to-market* aeg ning suureneb arendusprotsessi turvalisus, efektiivsus ja kvaliteet.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 34 leheküljel, 3 peatükki, 4 joonist.

# Abstract

## Adapting DevOps principles in Agile Development Process in Company Elisa Eesti AS

The goal of this research paper was to analyse the implementation of a DevOps development process in Elisa Eesti AS which is leading telecommunications company in Estonia. The research focused on the finding needed tools what must take into the use and changes to the current software architecture to implement DevOps principles in Elisa. As conclusions, the TO BE updates to the development process have been introduced, that will guarantee a more efficient development process. Some key changes have come out of this research, the utilization of which would allow Elisa to achieve a more efficient TO BE development process and to reduce time-to-market time and increase work quality. Firstly, in the TO BE process, the microservices architecture principles must be used when software is created or rewritten. Secondly there are necessity to implement software in Docker containers which allows to deploy software in every environment under the same circumstances. Thirdly, the infrastructure must be managed by the developers and the code. For that is recommended the OpenStack software which is Infrastructure as a Service (IaaS) cloud platform. Fourthly, the information system needs container orchestration tool to manage all the dependencies and connections between software services to automatically scale the system and recover from possible errors. In this paper Kubernetes and OpenShift are discussed for that functionality. All of these changes and tools will help to achieve a more efficient development process and reduce time-to-market time in company Elisa Eesti AS.

The thesis is in Estonian and contains 34 pages of text, 3 chapters, 4 figures.

# Lühendite ja mõistete sõnastik

| | |
|---|---|
| REST | *Representational state transfer,* tarkvaraarhitektuuri laad |
| HTTP(S) | *Hypertext Transfer Protocol (Secure)*, (turvaline) hüperteksti edastusprotokoll |
| SDLC | *Software Development Life Cycle*, Tarkvara arendusprotsess |
| AS IS | Praegune olukord |
| TO BE | Tuleviku olukord |
| IaaS | *Infrastructure as a service*, Taristu kui teenus |

# Sisukord

Sisukord ... 6
Jooniste loetelu ... 7
Sissejuhatus ... 8
1 DevOps arendusmetoodika ... 11
1.1 Agiilse arendusmetoodika ja DevOps põhimõtete erinevused ... 11
1.2 DevOps juurutamiseks vajalikud tööriistad ja põhimõtted ... 13
1.2.1 Mikroteenusarhitektuur ... 13
1.2.2 Docker ... 14
1.2.3 Taristu kui teenus (IaaS) ... 17
1.2.4 Süsteemi orkestratsioon ... 18
1.3 DevOps arhitektuur ja tööriistad Elisa Oyj's ... 18
2 Meetod ja valim ... 21
3 Devops arendusmetoodika juurutamise analüüs ... 21
3.1 Ülevaade AS IS olukorrast ... 22
3.2 AS IS arendusprotsess (SDLC) Elisas ... 23
3.3 TO BE arendusprotsess (SDLC) protsess Elisas ... 24
3.4 Tarkvara arhitektuur AS IS ja TO BE protsessis ... 25
3.5 Planeerimine AS IS ja TO BE protsessis ... 27
3.6 Kodeerimine AS IS ja TO BE protsessis ... 27
3.7 Tarkvara ehitamine AS IS ja TO BE protsessis ... 28
3.8 Automaattestide käivitamine AS IS ja TO BE protsessis ... 28
3.9 Testkeskkonda üles seadmine AS IS ja TO BE protsessis ... 29
3.10 Toodangu keskkonda üles seadmine AS IS ja TO BE protsessis ... 29
3.11 Jälgimine AS IS ja TO BE protsessis ... 30
4 Kokkuvõte ... 32
Kasutatud kirjandus ... 33

# Jooniste loetelu

Joonis 1. Tarkvara arenduse protsess (SDLC) ja agiilne osa selles [5]. ......................... 12
Joonis 2. Tarkvara arenduse protsess (SDLC) ja DevOps [5]. ...................................... 12
Joonis 3. Docker arhitektuur [11]. ....................................................................... 15
Joonis 4. Elisa Oyj. DevOps platvormi areng [19] ......................................................... 19

# Sissejuhatus

Uurimistöö objektiks on telekommunikatsiooniettevõtte Elisa Eesti AS infosüsteemide arendusprotsess ja selle parendamine. Elisa on enam kui 1000 töötajaga suurettevõte, mille käive oli 2017. aastal 157,9 miljonit eurot. Elisa on suurim erakliendi telekomi- ja TV-teenuste pakkuja ning suuruselt teine interneti püsiühenduse pakkuja Eesti turul [1]. Elisas arendatakse majasiseselt nii suuremaid, väiksemaid kui ka kõrge ja madala riskiga projekte.

Ettevõtetele on järjest enam oluline pakkuda kvaliteetset teenust ning tagada kliendirahulolu. Turg ja tehnoloogia on pidevas muutumises ning sellest tulenevalt muutuvad ka nõudmised juba loodud infosüsteemidele. Tänu sellele on ka tarkvaraarendus ja viis, kuidas seda teostatakse pidevas muutumises. Kliendid ootavad pidevat arengut ning kiiret reageerimist tagasisidele [2]. Paljud ettevõtted, sealhulgas ka Elisa, on teinud aastate jooksul väga palju muudatusi ja parendusi, et kogu arendusprotsess oleks võimalikult agiilne. Tänu sellele suudavad arendustiimid reageerida ärinõuete muutumisele oluliselt kiiremini ning suhtlus tellijate ning arendajate vahel on väga heal tasemel. Erinevad samaaegsed arendused liidetakse omavahel kokku automaatselt (*Continuous Integration*) ning arendajate omavaheline suhtlus toimub pidevalt. Kui agiilne tarkvaraarendus keskendub tellijate ja arendajate vahelisele suhtlusele ja kiirele tarkvaraarendusele, siis tähelepanuta on jäänud tarkvara arendajate ja tarkvara haldajate vaheline suhtlus ja protsesside parendamine. Sisuliselt tähendab see seda, et arendused saavad küll kiiresti valmis, aga nende kasutusele võtmine ehk tarnimine kliendile on kohmakas ja aeganõudev.

"DevOps" arendusmetoodika on kogum tavasid, mis peaksid vähendama aega arenduse valmimise ja tarnimise vahel seejuures kaotamata tarkvara kvaliteedis. Nende põhimõtete järgi peaks olema tarkvara tarnitavas seisus pidevalt ning tarnimine peaks toimuma vahetult peale selle valmimist ehk automaatselt (*Continuous Deployment*) [2]. Selline protsess eeldab tarkvaraga tegelevatelt ettevõtetelt mitmeid põhimõttelisi muudatusi ning väga head plaani, kuidas selleni jõuda.

Töö uurimisprobleem seisneb selles, et Elisas puudub plaan ja arusaamine kuidas juurutada „DevOps“ põhimõtted tänases arendusprotsessis. On jõutud arusaamiseni, mis on DevOps põhimõtted ja eesmärk, kuid kuna praktilisi näiteid nende juurutamisest on vähe, puuduvad rakenduslikud soovitused põhimõtete kasutusele võtmiseks. Lähtuvalt uurimisprobleemist on magistritöö eesmärk välja töötada protsessimuudatuste ettepanekud ning kirjeldada, millised tööriistad tuleks kasutusele võtta, et juurutada DevOps põhimõtted Elisa Eesti AS’is.

Magistritöö keksed uurimisküsimused on järgmised:

- Millised infotehnoloogilised tööriistad tuleb kasutusele võtta, et jõuda „*Continuous Integration*“ protsessist „*Continuous Deployment*“ protsessini?
- Millised muudatused tuleb teha tarkvara arhitektuuris, et jõuda „*Continuous Integration*“ protsessist „*Continuous Deployment*“ protsessini?

Lisaks on töös püstitatud järgmine alauurimisküsimus:

- Kuidas mõjutab DevOps arendusmetoodika tarkvara arendusprotsessi (SDLC) igat etappi?

Käesoleva uurimistöö puhul on tegemist kvalitatiivuuringuga. Täpsemalt on tegemist juhtumuuringuga ja kasutatud on ka võrdlevat analüüsimeetodit. Uurimisstrateegia põhineb induktiivse ja deduktiivse lähenemise kombineerimisel ehk töö raames kogutud empiiriliste ja teoreetiliste andmete põhjal on koostatud töö analüüsiraamistik. Empiiriline andmestik on kogutud poolstruktureeritud intervjuudega ning kasutatud on sihiteadlikku valimit.

Mõiste AS IS defineerib Elisas seni kasutusel olnud arendusprotsessi (Agiilne Scrum meetod). Mõiste TO BE defineerib Elisas juurutatavat DevOps põhimõtetel loodud arendusprotsessi. Teooria osas on avatud mõisted nagu mikroteenusarhitektuur, Docker, taristu kui platvorm ja konteinerite orkestratsioon. Lisaks on teooria osas käsitletud arendusprotsessi üldist tausta ning agiilse ning DevOps arendusprotsessi erinevusi. Töö analüüsi osa keskendub analüüsiraamistikus välja toodud mõjutegurite analüüsimisele. Selgitab välja, miks ühed või teised tegurid AS IS arendusprotsessis tingisid ebaefektiivsust ja kuidas DevOps arendusprotsessiga kaasnevad muutused võimaldavad

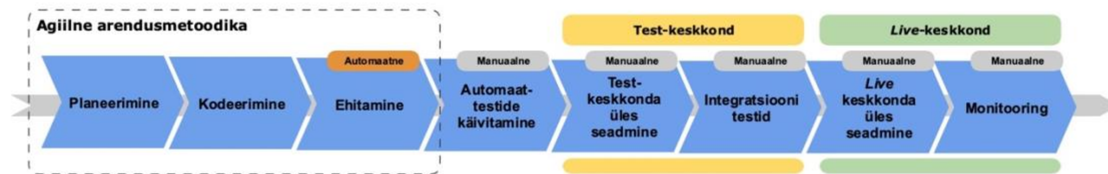
tagada tõhusama arendusprotsessi. Analüüsi alapeatükid keskenduvad tarkvara arhitektuuri, planeerimise, kodeerimise, automaattestide käivitamise, testkeskkonda paigaldamise, toodangu keskkonda paigaldamise ja jälgimise erinevustele AS IS ja TO BE arendusprotsessis.

# 1 DevOps arendusmetoodika

DevOps on saanud viimasel ajal väga palju tähelepanu, kui uut moodi mõtlemine, kus on ühendatud tarkvara arendus (*Development*) ja haldus (*Operations*). Kuigi DevOps mõistest ja eesmärkidest on palju räägitud, siis ühist arusaama, mida see endaga kaasa toob, pole saavutatud. Veel enam, paljud definitsioonid ja selgitused väljendavad vaid osaliselt DevOps põhimõtteid ning ei ole täpsed [3], [4]. Üks täpsemaid ja põhjalikumalt uuritud definitsioone, mille R. Jabbari, N. bin Ali, K. Petersen ja B. Tanveer oma uurimistöös „What is DevOps?: A Systematic Mapping Study on Definitions and Practices“ välja toovad, kõlab järgmiselt: *„*DevOps on arendusmetoodika, mille eesmärk on vähendada lõhet tarkvara arenduse ja halduse vahel, rõhutades suhtlust ja koostööd, pidevat integratsiooni, kvaliteedi tagamist ja automaatset tarkvara tarnet, kasutades selleks mitmeid arendustavasid. [3]“ On leitud, et DevOps põhimõtete juurutamine vähendab *time to market* aega, võimaldab kiiret ja pidevat tagasisidet, tasakaalustab kulu ja kvaliteedi suhet, annab tarkvara tarnele parema ennustatavuse ning tõstab kogu ettevõtte efektiivsust tervikuna [2].

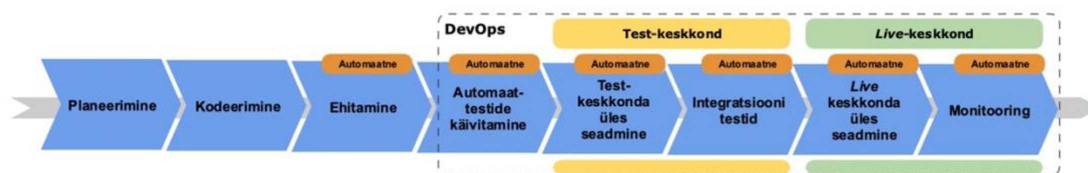
## 1.1 Agiilse arendusmetoodika ja DevOps põhimõtete erinevused

Agiilne arendusmetoodika lähtub põhimõttest, et tarkvara arendatakse iteratiivselt ehk kiht kihi haaval. Keskendutakse tellija ja arendustiimi vahelisele suhtlusele ning iga lühikese perioodi järel (*sprint*) näidatakse tellijale juba valminud tarkvara ning seejärel planeeritakse koos tellijaga järgmised arendused. See tähendab, et kogu tarkvara arenduse protsessist (SDLC), mis koosneb planeerimisest, kodeerimisest, tarkvara ehitamisest, automaattestide käivitamisest, testkeskkonda üles seadmisest, integratsiooni testide käivitamisest, toodangu (*Live*) keskkonda üles seadmisest ja jälgimisest, keskendutakse esimesele kolmele sammule (Joonis 1) [5]. Tänu sellele on tarkvara ehitamine ja integreerimine olemasoleva koodibaasiga tihtipeale automatiseeritud (*Continuous Integration*), kuid tarkvara arenduse protsessi ülejäänud sammud tehakse manuaalselt ja vastavalt vajadusele.




Joonis 1. Tarkvara arenduse protsess (SDLC) ja agiilne osa selles [5].

DevOps arendusmetoodika keskendub sellele, et valminud tarkvara jõuaks võimalikult kiiresti ja kvaliteetselt toodangu keskkonda, ehk lõppkasutajani. See tähendab, et kõik või suur osa tarkvara arenduse protsessist peab olema automatiseeritud [3]. Vahetult peale kodeerimist käivitatakse automaattestid ning peale nende edukat läbimist jõuavad muudatused automaatselt testkeskkonda (*Continuous Delivery*), kus omakorda käivituvad integratsioonitestid. Lisaks sellele on automatiseeritud ka toodangu (*Live*) keskkonda üles seadmine (*Continuous Deployment)* ning monitooring ehk jälgimine (Joonis 2) [5]. Sellise automatiseeritud protsessi saavutamiseks on vaja juurutada mitmeid erinevaid tööriistu ning muuta olemasolevaid põhimõtteid, kuidas tarkvara luua, testida ja hallata [2].




Joonis 2. Tarkvara arenduse protsess (SDLC) ja DevOps [5].

Agiilne arendusmetoodika ja DevOps arendusmetoodika on sama mündi erinevad pooled. Mõlemad tegelevad tarkvara arenduse protsessi parendamisega. Agiilne metoodika keskendub protsessi esimestele sammudele ning suhtlusele tellija ja arendaja vahel. DevOps seevastu protsessi viimastele sammudele ning suhtlusele arendaja ja haldaja vahel.

## 1.2 DevOps juurutamiseks vajalikud tööriistad ja põhimõtted

Automaatne tarkvara väljalase (*Continuous Deploy*) eeldab, et kasutusele on võetud infosüsteemile sobiv automaatne töövoog, mida rakendatakse peale igat muudatust ja mille eesmärk liigutada muudatus arendaja arvutist test- ja toodangukeskkonda. Töövoogude täide viimist on võimalik teostada väga erinevalt, kuid seni on end kõige edukamalt tõestanud Docker konteineritele baseeruvad lahendused, kus kasutatakse mikroteenusarhitektuuri koos dünaamilise konfiguratsioonihalduse ja teenuste orkestratsiooniga [6].

### 1.2.1 Mikroteenusarhitektuur

Väga paljud infosüsteemid on loodud monoliitse arhitektuuriga, kus üks rakendus täidab väga paljusid erinevaid funktsioone. Selline lähenemine sobib hästi infosüsteemidele, mida arendab korraga vähe arendajaid ning mille funktsionaalsus on piiratud. Mida suuremaks kasvab funktsionaalsus ning mida rohkem on süsteemi muutvaid osapooli, seda keerulisem on tagada rakenduse kiiret muudetavust ning muudatuste lühikest *time-to-market* aega. Seda probleemi on asutud lahendama mikroteenusarhitektuuriga [6].

„Mikroteenus (µT) on iseseisva elutsükliga, kiiresti arendatav, selgepiiriliste liidestega, iseseisev, ühte kasulikku funktsiooni täitev rakendus [7]." Iseseisev elutsükkel ja iseseisvus tähendab, et seda ei ole vaja arendada teiste süsteemiosadega samaaegselt ning ta sõltub võimalikult vähestest teistest mikroteenustest, teekidest, tehnoloogiatest ja raamistikest. Kiiresti arendatav tähendab, et teenuse muutmine või uuesti arendamine ei võta rohkem aega kui 1 nädal. See omadus on väga oluline, sest annab sõltumatuse programmeerimiskeeltest ja teostusviisidest ning vajadusel on võimalik olemasolev komponent hüljata ning kiiresti uuesti luua. Liidesed peavad olema lihtsad ning kompaktsed. Tänapäeval on selleks REST stiilis HTTP(S) protokoll, kus vahetatakse JSON formaadis andmeid [7], [6].

Mikroteenusarhitektuuriga kaasneb ka mitmeid väljakutseid, millega tuleb kindlasti tegeleda [6], [8], [9]:

- **Vigade isoleerimine.** Kuna süsteemi muudetakse tihti, siis võimalikud vead peavad välja tulema kiiresti ning nende põhjused ja asukoht selguma vahetult peale muudatuse tegemist. Lisaks sellele ei tohi ühe teenuse muudatus ja viga selles mõjutada teisi teenuseid.

- **Jälgitavus**. Ühe suure süsteemi asendamine paljude väikestega tekitab vajaduse visualiseerida kõigi süsteemiosade seisundit, et hinnata kogu infosüsteemi toimivust. See hõlmab endas ka tsentraliseeritud logimise lahendust, kuhu kõik teenused saavad logisid kirjutada ja salvestada ning kust on neid võimalik lihtsasti otsida.

- **Kohustus automatiseerida**. Mikroteenuste plahvatuslik kasv ja nende vahelised seosed sunnivad paljusid tegevusi, mida monoliitse süsteemi puhul oli võimalik teha manuaalselt, automatiseerima.

- **Sõltumatus**. Erinevate süsteemiosade lahti sidumine ning üksteisest sõltumatuna hoidmine ei ole lihtne, kuid see on kriitilise tähtsusega, et igat mikroteenust saaks iseseisvalt arendada ja hallata.

- **Testimine**. Mida rohkem on süsteemis erinevaid osasid, seda keerulisem on seda manuaalselt testida. Iga üksiku teenuse eraldi testimine ei anna kindlust, et kogu infosüsteem töötab nii nagu soovitud. Seetõttu on järjest olulisem automaattestide loomine ning automaatne käivitamine peale igat muudatust. Kriitilise tähtsusega on ka piisav integratsioonitestide hulk.

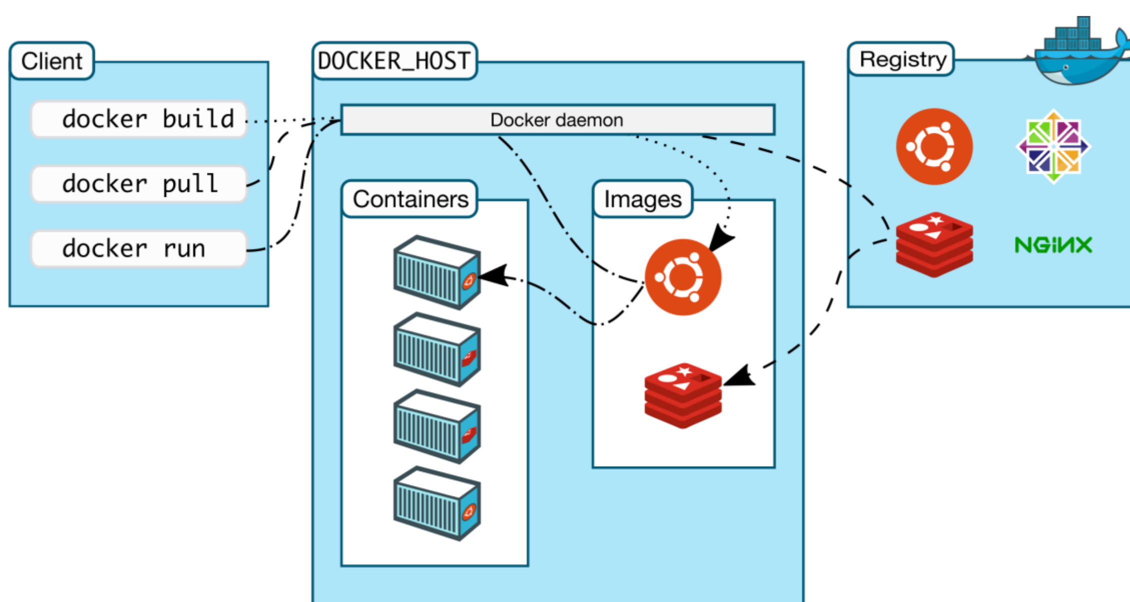
- **Laiendatavus**. Kuigi süsteemi lihtne mastaapsuse muutmine on peamine aspekt, miks luua infosüsteem kasutades mikroteenuseid, siis laiendatavuse saavutamine eeldab mitmete detailide peale mõtlemist. Kuna teenuseid on väga palju ja ühendusi nende vahel veel rohkem, siis on vaja erilahendusi nende ühenduste juhtimiseks võrgus, et võrguliiklus liiguks kindlasti õigete rakendusliidesteni (API). Lisaks peab olema süsteemikonfiguratsioon automaatselt ja lihtsasti muudetav. See eeldab omakorda dünaamilisemat ja keerukamat konfiguratsioonihaldust ja orkestratsiooni.

### 1.2.2 Docker

Docker on virtualiseerimise tehnoloogia, mis võimaldab rakenduse ja tema sõltuvused ümbritseda konteineriga, mida on võimalik lihtsasti paigaldada väga erinevatele platvormidele, alates arendaja arvutist kuni ettevõtte serveriteni ja pilveteenuste pakkujateni [10]. Konteinerite kasuks räägib see, et üks infosüsteem saab koosneda väga paljudest erinevatest tükkidest, mis asuvad igaüks erinevas ja üksteisest sõltumatus

konteineris, mida on seetõttu võimalik eraldiseisvalt arendada ja hallata [6]. Lisaks on võimalik väga lihtsasti muuta süsteemi jõudlust, kui sama sisuga konteinerite arvu suurendada või vähendada. See eeldab aga selleks sobivat rakenduse teostust ning arhitektuuri.

Docker on oma olemuselt klient-server lahendus, mis koosneb konteineritest (*Containers*), kujutistest (*Images*), Docker deemonist (*Docker daemon*) ja kujutiste registrist (*Registry*) (Joonis 3) [11]. Deemoni kaudu on võimalik kliendil juhtida läbi REST API konteinerite, kujutiste, võrguressursside ja salvestusmeedia omavahelist suhtlust ning käitumist. Dockeri kujutisi on võimalik luua võttes aluseks juba varem loodud kujutis. Sisuliselt see tähendab, et kujutis koostatakse kiht kihi haaval. Ühes kujutises on tavaliselt aluskujutiseks operatsioonisüsteem ning selle peal omakorda kasutaja poolt soovitud kujutised, mida rakendus töötamiseks vajab. Kõige viimase kihina on kasutaja loodud rakendus. Kõik rakenduse jaoks vajalikud sõltuvused asuvad ühes kujutises ning sellest luuakse konteiner, mida on võimalik majutada väga erinevates kohtades. Kui kujutises muudetakse mingit kihti, näiteks operatsioonisüsteemi, siis ülejäänud kihte ei muudeta ning luuakse uus kujutis. Seetõttu on uute kujutiste loomine väga lihtne ning kiire [11], [6].




Joonis 3. Docker arhitektuur [11].

Mikroteenusarhitektuur ja Dockeri konteinerid sobivad omavahel väga hästi, sest Docker aitab lahendada mitmeid väljakutseid, mille mikroteenusarhitektuur põhjustab [12], [6]:

- **Kiirendab automatiseerimist.** Kõigi rakenduste ja teenuste käivitamise skriptid ja konfiguratsioon asub konteinerite sees, aga väliselt käituvad kõik konteinerid väga sarnaselt. Selle tõttu on väga lihtne erinevaid konteinereid tarkvara arenduse protsessis samadel alustel kohelda, mis omakorda võimaldab erinevaid samme (integratsioonitestide käivitamine, test- või toodangukeskkonda paigaldamine jne.) lihtsasti automatiseerida.

- **Suurendab iseseisvust.** Iga konteiner selles asuva teenusega on täiesti iseseisev ning tänu sellele on võimalik seda teistest sõltumatult arendada ning hallata. Lisaks suurendavad konteinerid võimalust, et iga teenus on arendatud arendustiimile kõige sobivamas programmeerimiskeeles ja tööriistu kasutades.

- **Lihtsustab teisaldatavust.** Iga mikroteenus on võimalik paigaldada omaette konteinerisse ning konteinerite paigaldamine erinevatele platvormidele on oluliselt lihtsam, kui mikroteenuste ükshaaval paigaldamine erinevatesse keskkondadesse. See tähendab, et erinevad huvigrupid nagu tellijad, arendajad, testijad ja haldajad saavad sama konteineri paigaldada vastavalt oma soovile, kas enda arvutisse, virtuaalserverisse, füüsilisse serverisse või pilve ning nad saavad kindlad olla, et igas keskkonnas käitub rakendus täpselt samamoodi, mitte ei olene serverist ja selle konfiguratsioonist.

- **Annab parema ressursikasutuse.** Dockeri puhul on igas konteineris täpselt nii palju sõltuvusi, kui palju teenuse toimimise jaoks vaja on. Mitte rohkem ega vähem. Kuna konteinereid on iseseisvad üksused ja saavad jagada sama operatsioonisüsteemi kernelit siis on võimalik neid ühte serverisse paigaldada mitmeid ning ressursikasutus on efektiivsem. Kuna konteinerite sisu on optimaalne on neid võimalik ühte füüsilisse serverisse paigutada rohkem, kui virtuaalservereid.

- **Täiendav turvalisus.** Rakenduse sensitiivne konfiguratsioon asub konteineri sees ning ei ole väljast poolt vaadeldav. Lisaks on võimalik igas arenduse sammus rakendada konteineritele lihtsamalt erinevaid automaateid turvateste.

### 1.2.3 Taristu kui teenus (IaaS)

Tarkvara väljalaske üks oluline etapp on tarkvara jaoks vajaliku taristu ehk riistvara seadistamine ja üles seadmine. Väga tihti on see manuaalne protsess ning olenevalt arendajate ja serveriadministraatorite vahelisest koostööst võib see võtta aega päevadest kuni nädalateni. Seoses sooviga tarkvara pidevalt ja automaatselt tarnida on see oluline takistus. Seetõttu kasutavad arendusfirmad järjest rohkem pilvelahendusi, kus riistvara ja selle seadistamisega ei ole vaja ise tegeleda (IaaS) [2]. Selliste andmekeskuste kasutamine, kus füüsiliste ressursside haldamiseks, jälgimiseks ja juhtimiseks kasutatakse CloudStack, Eucalyptus, OpenStack või samaväärset haldustarkvara, annab arendajatele võimaluse juhtida riistavara lähtekoodiga ning seda nimetatakse „taristu kui kood“ lahenduseks (IaaC) [6].

Kõige laiemalt levinud pilveandmekeskuste haldustarkvara on OpenStack. OpenStack on avatud lähtekoodiga tarkvara, mille arendamist alustas NASA ja Rackspace ning mida on võimalik kasutada nii privaatsete kui ka avalike pilvelahenduste haldamiseks [13]. See koosneb omakorda väga mitmetest erinevatest avatud lähtekoodiga projektidest ning tänapäeval arendab OpenStack platvormi Red Hat nimeline ettevõte [14]. Näiteks kasutatakse OpenStack haldustarkvaras füüsiliste serverite arvutusvõimsuse juhtimiseks Nova ja Glance nimelisi tööriistu. Võrguliikluse juhtimiseks on Neutron, kõvaketaste kasutamiseks Swift ja Cinder. Lisaks kasutatakse ka väga erinevaid tööriistu, et kõikvõimalikud erinevate rakenduste poolt soovitud funktsionaalsused oleksid kaetud [13].

Võrreldes tavapärase füüsilise serveri virtualiseerimisega, mida on võimalik saavutada näiteks VMware nimelise tarkvaraga, annab OpenStack ehk „taristu kui teenus“ (IaaS) tüüpi lähenemine DevOps arendajatele olulise eelise. Senine manuaalne töö, mida teevad serveri administraatorid, on võimalik automatiseerida. Sisuliselt kaob administraatorite roll arenduste toodangusse tarnimise etapis ära ning arendajad saavad seda ise automaatselt juhtida. Kui ettevõte võtab kasutusele privaatse andmetöötluspilve, siis administraatorite rolliks jääb ainult selle üldine tõrgeteta töö tagamine, mitte uute serverite loomine vastavalt arendusvajadustele [13] [8].

### 1.2.4 Süsteemi orkestratsioon

Mikroteenusarhitektuuri alusel loodud infosüsteem koosneb mitmetest klastritest, milledes võib olla sadu erinevaid teenuseid, mis asuvad omakorda erinevates Docker konteinerites [15]. Mida suuremaks taolised süsteemid kasvavad, seda keerulisem on kõigi nende konteinerite omavahelisi seoseid ja tarnimist juhtida. Et seda probleemi lahendada on vaja võtta kasutusele konteinerite orkestratsiooni platvorm, mille abil on võimalik automaatselt juhtida rakenduste tarnimist, mastaapsuse muutmist ja konteinerite elukaart [15], [16]. Orkestratsiooniplatvorme on olemas väga erinevaid, kuid hetkel on kõige populaarsem Kubernetes nimeline tarkvara, sest see toetab väga erinevaid alusplatvorme, olles seejuures lihtsasti konfigureeritav ja liigendatav [16].

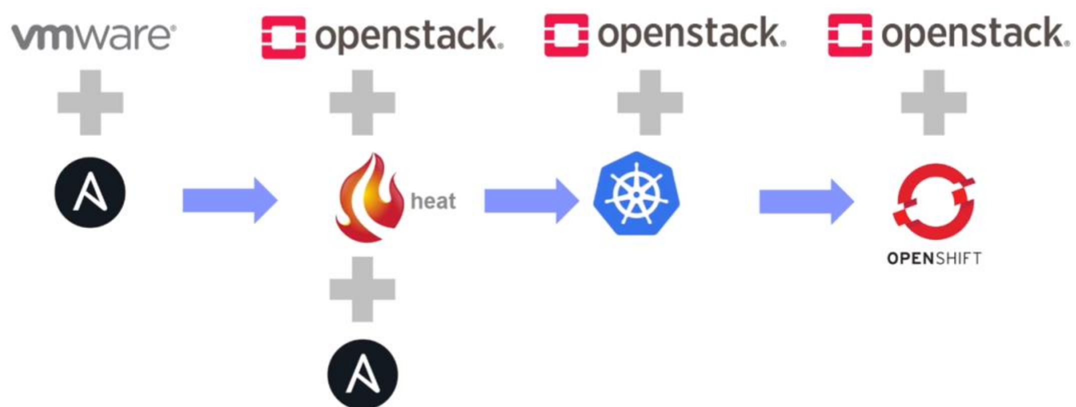
Kubernetes platvormil on kõige väiksemaks konfigureeritavaks objektiks *Pod*, mis koosneb ühest või mitmest konteinerist. Iga *Pod'*i on võimalik konfigureerida, kui palju ta minimaalselt ja maksimaalselt serveri ressurssi vajab. *Pod*'id asuvad erinevates Kubernetese sõlmedes (*Node*), mida omakorda juhib *Master*, kes kontrollib, et iga teenus töötaks nii nagu peab ning, et logid jõuaksid ühte kohta [17]. Oluline funktsionaalsus on automaatne veaparandus ning mastaapsuse suurendamine. See tähendab, et kui üks *Pod* või sõlm lõpetab oma töö veaga, suudab *Master* käivitada uue *Pod*'i või sõlme automaatselt.

## 1.3 DevOps arhitektuur ja tööriistad Elisa Oyj's

Elisa Oyj. on Soome suurim telekomi- ja digitaalteenuste pakkuja ning Elisa Eesti AS emafirma, kellega koos pakutakse teenuseid Soomes ja Eestis 2,8 miljonile kliendile. Kogu oma 136 aastase ajaloo vältel on Elisa kasutanud uudseimat tehnoloogiat ning nutikaid viise töö tegemiseks [18]. Nii Elisa Oyj. kui Elisa Eesti AS arendavad oma tarkvara suures osas ise, kuid üksteisest praktiliselt sõltumatult. Magistritöö autor on kohtunud Elisa Oyj. DevOps arendus- ja haldustiimiga korduvalt, et uurida millised tööriistad ja arhitektuur on seal kasutusele võetud DevOps arendusmetoodika realiseerimiseks.

Elisa Oyj. DevOps meeskonna ülesanne on pakkuda kõigile ettevõtte arendustiimidele privaatset taristu kui teenus (Iaas) lahendust, et arendajad saaksid ise oma tarkvara arendada ja hallata. Nimetatud DevOps meeskond alustas arendajate toetamist tehnoloogiatega VMware ja Ansible, millest esimesega virtualiseeritakse füüsilisi

servereid ning teisega lihtsustatakse tarkvara tarnimist nendele serveritele. Kuna VMware ei võimalda taristut vajalikul määral hallata, siis võeti kasutusele OpenStack nimeline serverite haldustarkvara koos selles sisalduva Heat tööriistaga, millega oli võimalik erinevaid rakendusi orkestreerida ning alles jäi Ansible, millega tarniti rakendusi erinevatesse keskkondadesse. Järgmise sammuna asendati Heat ja Ansible Kubernetes nimelise konteinerite orkestreerimise tööriistaga, sest paljud arendajad olid kasutusele võtnud Docker konteinerid tarkvara tarnimiseks. Kubernetese kasutamise tulemusel selgus, et antud lahendusega pole võimalik saavutada olukorda, kus samal platvormil asuvad erinevate arendustiimide rakendused isoleeritult ning seetõttu oli vaja platvormile installeerida palju erinevaid Kubernetese instantse. Selle probleemi lahendamiseks võeti kasutusele OpenShift, mis kasutab sisemiselt Kubernetes tarkvara, kuid lisab sellele mitmeid täiendavaid tööriistu, sealhulgas rakenduste ja erinevate keskkondade isoleerimise võimekuse [19]. Kogu platvormil kasutatud tehnoloogiate areng on välja toodud ka Joonisel 4.




Joonis 4. Elisa Oyj. DevOps platvormi areng [19]

Peamised eesmärgid, mille poole Elisa Oyj. püüdleb DevOps arendusmetoodika ja kasutusele võetud arhitektuuriga on [19]:

- **Tarkvarapõhine pilvelahendus**, mis tagab kiire, efektiivse ja paindliku teenuste arenduse.
- **Kiirem õppimise- ja arendusprotsess**, tänu sagedasema ja automaatse tarkvara tarne võimekusele.

- **Vastutus ja võimekus lõppkasutaja kasutuskogemuse osas on ainult arendustiimidel.** Ei ole enam jagatud vastutust tarkvara haldajate ja arendajate vahel. Kogu vastutus on ainult arendustiimil, sest nad saavad ka riistvara juhtida.

# 2 Meetod ja valim

Käesolevas magistritöös on tegemist kvalitatiivuuringuga. Täpsemalt, kuna uurimisfookuses on ettevõttes Elisa Eesti AS DevOps arendusmetoodika juurutamine, on tegemist juhtumuuringuga. Samuti on töö analüüsi osas kasutatud võrdlevat analüüsimeetodit. Uut, loodavat (TO BE) arendusmetoodikat on võrreldud seni kehtiva (AS IS) arendusmetoodikaga, et analüüsida protsessi muutmist tinginud tegureid ja TO BE protsessi potentsiaalselt efektiivsust.

DevOps arendusmetoodika juurutamise uurimine on oluline, kuna IT maailmas on järjest enam hakatud agiilset arendusprotsessi täiendama DevOps arendusmetoodikaga. Selle põhjuseks on asjaolu, et DevOps arendusmetoodika võimaldab sageli suurendada töö tõhusust ehk vähendada arendusnõuete *time-to-market* aega ja tagab tõenäolisemalt süsteemi kiire muudetavuse ning suurema kvaliteedi.

Uurimisstrateegia põhineb deduktiivse ja induktiivse lähenemise kombineerimisel. Autor on koondanud teoreetilise ja empiirilise andmestiku ning selle põhjal konstrueerinud analüüsiraamistiku, milles sisalduvaid mõjutegureid töös analüüsitakse.

Empiiriliste andmete kogumiseks on kasutatud poolstruktureeritud intervjuu meetodit ja tegemist on sihiteadliku valimiga. Töö raames on intervjueeritud kolme Elisa Eesti AS töötajat, kes on seotud tarkvara arhitektuuri ning arendusega. Lisaks on autor kasutanud enda teadmisi Elisa infosüsteemidest ja protsessidest, olles saanud kogemuse rohkem kui 9 aasta jooksul, töötades Elisas Product Owneri, süsteemianalüütiku, arendaja, tarkvara arhitekti ja arendajate tiimijuhi ametikohtadel.

# 3 Devops arendusmetoodika juurutamise analüüs

Analüüs keskendub teguritele, mis muutuvad erinevates tarkvara arendusprotsessi (SDLC) etappides, kui kasutusele võetakse Elisa Oyj. DevOps tiimi poolt loodud privaatne pilvelahendus. Lisaks vaadeldakse, kuidas tuleb muuta tarkvara arhitektuuri, et see sobituks antud pilvelahendusega. Muudatuse üldiseks ajendiks on vajadus tagada tulemuslikum ning ressursse mitte raiskav töö, nii arendusnõuete realiseerimisel kui

hilisemal haldamisel. Seega keskendub analüüs esmalt seni kehtiva (AS IS) protsessi olulisematele probleemkohtadele, teisalt on aga analüüsi fookuses võtmemuudatused, mida tuleks tõhusama arendusprotsessi tagamiseks juurutada.

Empiirilised andmed on kogutud poolstruktureeritud intervjuudega ja intervjueeritavad on valitud sihiteadliku valimina. Intervjuu 1 [20] on läbi viidud Elisa tarkvara arhitektiga, kes on ühtlasi ka mentor uutele arendajatele. Tema roll arendusprotsessi muutmisel on otsustada millised muudatused läbi viiakse ja kuidas need realiseeritakse. Intervjuu 2 [21] on läbi viidud Elisa teise tarkvara arhitektiga, kes on ka andmebaasi administraatori rollis. Tema roll arendusprotsessi muutmisel on sama esimese arhitektiga. Intervjuu 3 [22] on läbi viidud Elisa juhtivarendajaga, mis on arendajate kõrgeim karjääritase Elisas. Tema roll on lisaks arendustiimi töös osalemisele ka vastutata kasutajaliideste arhitektuuri eest. Autori roll arendusprotsessis on leida arendajate tiimijuhina parimad praktikad ja tehnoloogiad uute arenduste realiseerimiseks ja haldamiseks, et arendajate efektiivsus oleks maksimaalne. Lisaks on autor osalenud mitmetes töötubades koos Elisa Oyj. DevOps arendusmeeskonna ning teiste arendusmeeskondadega, kust on kogutud suur hulk empiirilisi andmeid.

## 3.1 Ülevaade AS IS olukorrast

Elisas arendatakse põhiliselt Elisa iseteenindusbürood, E-poodi ja kliendihaldussüsteemi. Kõige suurem ja keerukam on neist kliendihaldussüsteem, mis koosneb väga paljudest erinevatest suurematest ja väiksematest rakendustest ning mille põhilised funktsionaalsused on kliendiandmete haldamine, toodete ja teenuste haldamine ning võrgu- ning arveldussüsteemidesse info proviseerimine (*provisioning*). Igapäevaselt kasutavad kliendihaldussüsteeme sajad Elisa töötajad ning Iseteenindust ja E-poodi kümned tuhanded kliendid. Rakendused on valdavalt realiseeritud kasutades Java programmeerimise keelt ning andmebaasi haldamise süsteem on MariaDB. Kliendihaldussüsteemi kasutajaliidesed on loodud erinevate tehnoloogiatega nagu Freemarker, Wicket ja Angular JS. Iseteenindusbüroo ja E-pood kasutajaliidesed on realiseeritud Angular JS tehnoloogiaga ning lisaks sellele on kasutusele võetud mitmeid lisateeke, et lihtsustada ja optimeerida kasutajakogemust ning süsteemi arendatavust.

Kogu arendusprotsess on loodud agiilsetele põhimõtetele tuginedes ja arendussoovide uuesti prioriseerimine ja valideerimine toimub kogu protsessi vältel korduvalt. Äriüksuste

arendustellimuste prioriseerimist toetab kiire arendussoovide keerukuse hindamine. Sisuliselt tähendab see seda, et mitu korda nädalas kogunevad ühisele koosolekule tarkvara arhitektid ja teised spetsialistid ning hinnatakse kõigi uute arendussoovide keerukust. Peale seda koondatakse kõik arendustellimused ettevõtte-ülesesse prioriteetide nimekirja, kus kõrgemal asuvad need tellimused, millel on kõige suurem kulu ja tulu suhe. Seal nimekirjas kõige kõrgemal olevad tellimused liiguvad selle arendustiimi tööde järjekorda, kus see kõige kiiremini valmis saab. Sellest tulenevalt peavad kõik arendustiimid hakkama saama kõigi arendustellimustega ehk kasutusel on *end-to-end* arendamise loogika. See tähendab, et igal arendustiimil peab olema võimekus ükskõik millist süsteemiosa arendada.

Kõiki varem nimetatud rakendusi arendavad umbes 30 tarkvaraarendajat, kes jagunevad 6-te *Scrum*-tiimi. Igal tiimil on oma testija (QA), analüütik ning *ProductOwner* (PO). Analüütiku roll on arendustellimus tükeldada väiksemateks hinnatavateks ärilisteks osadeks, arvestades seniseid äritellimusi ja seoseid teiste süsteemidega. PO roll on esindada äritellijat ning jälgida, et kõik valminud arendused rahuldaksid ka tellijat. Lisaks toetavad kõiki arendustiime tarkvara arhitektid, kelle ülesanne on osaleda tiimi koosolekutel ning anda arendajatele juhiseid, kuidas kõiki arendustellimusi kõige optimaalsemalt on võimalik tehniliselt lahendada. Lisaks sellele on arhitektide roll jälgida pidevalt infotehnoloogia arenguid ning valida Elisa infosüsteemidele parimad tehnoloogiad ning tehnilised arengusuunad.

## 3.2 AS IS arendusprotsess (SDLC) Elisas

SDLC (*Software Development Life Cycle*) protsess, koosneb planeerimisest, kodeerimisest, tarkvara ehitamisest, automaattestide käivitamisest, testkeskkonda üles seadmisest, integratsiooni testide käivitamisest, toodangu (*Live*) keskkonda üles seadmisest ja jälgimisest.

Elisa infosüsteemide kood asub Git koodihoidlas ning arendusjärgus olev kood kirjutatakse *feature branch*'idesse, mis liidetakse põhiharru umbes nädal enne rakenduste toodangu keskkonda paigaldamist. Test keskkonna tarbeks liidetakse kood kokku pool-automaatselt Jenkins nimelise tööriistaga ning seal olevad rakendused ehitatakse ja paigaldatakse testkeskkonda pool-automaatselt Jenkinsi kasutajaliidese kaudu.

Rakendustes olevaid automaatteste käivitatakse samamoodi Jenkinsi kaudu. Integratsiooniteste teostatakse valdavalt manuaalselt testijate poolt.

Toodangu keskkonna tarbeks luuakse eraldiseisev Git'i haru, mida nimetatakse *release branch'*iks ning toodangu rakendused ehitatakse seal asuva koodi alusel. Lisaks on olemas veel prelive nimeline keskkond, kus asuvad kõige uuema *release* branch'i alusel ehitatud rakendused ning kus testitakse manuaalselt enne toodangu keskkonda tarnimist kõik rakendused üle.

Kui tegemist ei ole olemasoleva rakenduse muutmisega vaid täiesti uue rakenduse loomisega, siis tihtipeale on vajalik ka uute serverite tellimine taristu osakonnast. See võib võtta aega mõnest päevast kuni mõne nädalani, olenevalt süsteemi keerukusest ja serverite arvust.

Lisaks uue tarkvara loomisele ning tarnimisele tegelevad arendustiimid ka olemasoleva tarkvara haldamisega. Selleks tuleb kõigepealt luua tarkvarale erinevad jälgimis-skriptid, mis annavad märku, kui tarkvara ei käitu soovitud loogika alusel. Erinevatel rakendustel on need väga erinevad ning alati ei ole arendustiimil meeles neid luua. Sellisel juhul annavad süsteemi valest käitumisest teada selle kasutajad ning tagajärgedega tegelemine hõlmab endas tihti ka andmete parandamist.

Kogu protsessis on väga palju korduvaid tegevusi, mida tehakse manuaalselt, sealhulgas kvaliteedi tagamine, mille eest vastutab igas tiimis 1 inimene (QA). Praeguse protsessi alusel tarnitakse arendused toodangu keskkonda keskmiselt kord kahe nädala jooksul ning muudatuste mõju on pigem suur, sest korraga muutub süsteem olulisel määral.

## 3.3 TO BE arendusprotsess (SDLC) protsess Elisas

TO BE arendusprotsess (SDLC) on koostatud lähtudes arengutest, mis on toimunud infotehnoloogias. Aluseks on võetud DevOps arendusmetoodika põhimõtted ning erinevate ettevõtete ja uuringute tulemused nende juurutamisel, mis on välja toodud ka töö varasemates peatükkides. Sealhulgas Elisa Oyj. kogemus ja taristu arhitektuur. Tänu korduvatele kohtumistele on saavutatud kokkulepe, et Elisa Eesti arendustiimid saavad DevOps põhimõtete juurutamiseks kasutada Elisa Oyj. DevOps tiimi poolt loodud privaatset pilvelahendust, kus kasutatakse OpenStack ja OpenShift tehnoloogiat.

Pilvelahenduse kasutusele võtmine eeldab rakenduste sobitamist Docker konteineritega. Sisuliselt tähendab see seda, et mitmete rakenduste ülesehitust ja konfiguratsiooni tuleb olulisel määral muuta, et samast rakendusest saaks paralleelselt toimida mitu instantsi. Lisaks tuleb juurutada mikroteenusarhitektuuri põhimõtted ja rakenduste omavahelisi sõltuvusi vähendada. Olemasolevate rakenduste ümbermajutamine tähendab ka kogu võrguliikluse üle vaatamist ja tulemüüri reeglite ümberkonfigureerimist.

Planeerimise faasis tuleb arvestada esialgu suuremate töömahtudega, et uued arendused oleks võimalik luua kasutades mikroteenusarhitektuuri põhimõtteid ning rakendused töötaksid Docker konteinerites. See tähendab mitmete konfiguratsioonifailide loomist ning täiendavat testimist, et maandada muudatuste negatiivset mõju. Kui rakendused on loodud nii, et neid on võimalik tarnida konteinerites, siis saab nende paigaldamise erinevatesse keskkondadesse automatiseerida. Selleks tuleb planeerida ja teostada täiendav hulk tööd, kuid see on ühekordne.

Docker konteinerite kasutusele võtmine annab võimaluse erinevaid rakenduste versioone paremini hallata ning mikroteenusarhitektuuri kasutusele võtmine vähendab erinevate arenduste konflikte koodibaasis, sest arendused tarnitakse testkeskkonda vahetult peale koodi lisamist koodibaasi ja muudatuste mõju on seetõttu väiksem.

Esimeses TO BE protsessi versioonis on mõistlik liikuda kõigepealt *Continuous Delivery* automatiseerimise tasemeni. See tähendab, et valminud arendused tarnitakse automaatselt testkeskkonda. Sealt edasi liiguvad arendused toodangu keskkonda pool automaatselt kasutaja sekkumisel. Kogu arendusprotsessi automatiseerimine (*Continuous Delivery*) on plaanis juurutada peale seda, kui esimene versioon TO BE protsessist on edukalt juurutatud.

## 3.4 Tarkvara arhitektuur AS IS ja TO BE protsessis

Kuna Elisa arendatavad infosüsteemid koosnevad rohkem kui 30’st alamsüsteemist, siis antud magistritöö raames vaadeldakse detailselt ühe alamsüsteemi AS IS arhitektuuri ning analüüsitakse selle TO BE arhitektuuri, et arendamine saaks lähtuda DevOps põhimõtetest. Töös käsitletava süsteemi valik lähtub vajadusest suurendada rakenduse jõudlust ning seda on võimalik saavutada DevOps arendusmetoodika ja arhitektuuri rakendamisel.

Alamsüsteem, mille nimi on Mappo, tegeleb mobiilinumbrite võrgusüsteemidesse proviseerimisega. Kõik muudatused, nagu näiteks liitumine, lõpetamine ja paketivahetus, mida erinevates ärilistes protsessides mobiilinumbritega teostatakse, proviseeritakse Mappo poolt automaatselt võrku.

Mappo lähtekood on kirjutatud keeles Java ning rakendusel ei ole kasutajaliidest. Rakendus saab sisendi otse andmebaasitrigeritest, mis sisestavad selleks loodud andmebaasitabelisse info mobiilinumbrite kohta, mida on kliendihaldussüsteemis muudetud. Seda tabelit loeb Mappo pideva protsessina. Selle info alusel võrreldakse kliendihaldussüsteemi andmebaasis olevat informatsiooni Mappo enda andmebaasis oleva informatsiooniga ning leitakse erinevused. Erinevuste alusel luuakse vajalik xml formaadis käsk ning see edastatakse järgmisele infosüsteemile (Provi), mis tõlgib selle käsu sihtsüsteemile sobivasse protokolli. Sõnumite edastamine Mappo ja Provi vahel toimub asünkroonselt läbi sõnumite järjekorra (*queue*). Peale seda, kui sihtsüsteem on vastanud positiivse vastusega edastatakse see Provi poolt Mappole tagasi. Mappo kirjutab muudetud andmete kohta info enda andmebaasi ning võrdleb uuesti kliendihaldussüsteemis olevat informatsiooni enda andmebaasis oleva informatsiooniga. Tegevust korratakse seni, kuni erinevusi enam ei eksisteeri. Kui sihtsüsteem vastab veakoodiga või ei vasta üldse, siis kogu protsess selle mobiilinumbriga katkeb ning vajalik on manuaalne kasutajapoolne sekkumine, et leida vea põhjus ning see lahendada.

AS IS Mappo arhitektuuri plussideks on, et väliseid sõltuvusi on vähe ning rakendus töötab taustal ülejäänud infosüsteemist eraldiseisvalt. Miinuseks on piiratud jõudlus. Kuna igal mobiilinumbril on palju parameetreid, mis võivad muutuda, tuleb nende võrdlemiseks kahe andmebaasi vahel teha väga palju päringuid. Lisaks sellele pole rakendus loodud mitmes instantsis korraga käima. See tähendab, et kui rakendus konfigureerida töötama Docker konteineris ning tarnida OpenShift platvormile, siis selle jõudlust ei saa suurendada lisakonteineri tööle panemisega.

TO BE Mappo arhitektuur hõlmab endas rakenduse võimekust töötada samaaegselt mitmes sõlmpunktis (*node*). Selleks tuleb täiendada andmebaasitrigerite poolt loodud kirjete lugemise loogikat, et rakendus grupeeriks muudatuskirjed mobiilinumbripõhiselt ning töötlusesse võtmise ajal märgiks need enda nimele. Seda on vaja selleks, et rakenduse teised instantsid neid muudatusi töötlema ei asuks. Lisaks peab olema igal Mappo instantsil oma andmebaas ning need andmebaasid peavad olema omavahel

sünkroniseeritud. Selleks tuleks kasutusele võtta Galera klaster või samaväärne lahendus, kus andmebaasist on võimalik luua mitmeid instantse, kuid kogu sisu sünkroniseeritakse automaatselt teistes sõlmpunktides olevate andmebaasidega. Sellise arhitektuuriga on võimalik rakendusest ja selle andmebaasist teha vastavalt vajadusele nii palju koopiaid, kui on vaja ning sellega süsteemi jõudlust suurendada. Seejärel tuleb rakendus ja andmebaas konfigureerida tööle Docker konteineri sees töötavaks ning see on võimalik seada üles Elisa Oyj. rakenduste pilve.

Üldine arhitektuurne suunis kõigi rakenduste arendamisel on kasutada mikroteenusarhitektuuri põhimõtteid ning Docker konteinereid rakenduste tarnimisel. See annab võimaluse kõiki rakenduste arendamise ja haldamise etappe samadel alustel automatiseerida ning see lihtsustab oluliselt ka arendamist ja haldamist.

## 3.5 Planeerimine AS IS ja TO BE protsessis

Planeerimine AS IS protsessis toimub iteratiivselt. Arendustiim planeerib endale detailselt tööde hulga, mille ta on võimeline teostama kahe nädalaga. Kahe nädala pärast näidatakse valminud arendusi ning planeeritakse järgmisi töid. Detailse planeerimise aluseks on olemasoleva infosüsteemi hea tundmine ning mõistmine, kuidas arendussoovi süsteemis realiseerida.

TO BE protsessis muutub ainult planeeritava töö iseloom. Ülejäänud protsess jääb samaks. Tuleb arvestada olemasoleva süsteemi konfiguratsiooni muutmisega ja täiendavate teekide ja suhtlusviiside kasutusele võtmisega, et kõik rakendused saaksid töötada Docker konteinerites ja lähtuksid mikroteenusarhitektuuri põhimõtetest. See tähendab, et tööde mahud esialgu suurenevad.

## 3.6 Kodeerimine AS IS ja TO BE protsessis

AS IS protsessis alustatakse kodeerimist eraldiseisvasse Git *feature branch'*i, et teised arendustiimid oleksid võimalikult vähesel määral mõjutatud. See tuleneb asjaolust, et sama koodifaili võivad samal ajal muuta ka teised arendustiimid ja sellisel juhul tekib failis konflikt. Kui äriloogika on süsteemikoodina realiseeritud ning see on QA poolt testitud, lisatakse kood *feature branch*'ist põhiharru ja võimalikud konfliktid

lahendatakse selle käigus manuaalselt. Lisaks on suhteliselt jäigalt paigas tehnoloogiad, mida kodeerimisel tuleb kasutada.

TO BE protsessis on kasutusele võetud mikroteenusarhitektuur ning süsteemi erinevad osad suhtlevad omavahel REST API'ide kaudu. See tähendab, et iga väiksemat osa arendab erinev arendustiim ning koodi konflikte esineb vähem või üldse mitte. Sellisel juhul on võimalik loobuda *feature branch*'idest ning kõik arendustiimid saavad arendada koodi otse põhiharru. Lisaks kaob ära vajadus kasutada tehnoloogiaid, millega arendustiim ennast mugavalt ei tunne. Iga mikroteenus suhtleb teiste teenustega kokku lepitud API kaudu ning see pole enam oluline, millises tehnoloogias rakendus sisemiselt realiseeritud on.

## 3.7 Tarkvara ehitamine AS IS ja TO BE protsessis

AS IS protsessis kasutatakse väga erinevaid tehnoloogiaid ja mitmeid manuaalseid protsesse, et rakendusi ehitada. Põhiliselt kasutatakse Ant, Ivy, Maven ja Ansible nimelisi tehnoloogiaid, millel igal ühel on oma põhimõtted ja käivitamise loogika. See tähendab lisakeerukust uutele arendajatele süsteemi õppimisel ning konfiguratsioonifailide loomisel.

TO BE arendusprotsessis asuvad rakenduste konfiguratsioonid Docker failides ning kõigi rakenduste ehitamine ja sõltuvuste haldamine toimub sama loogika alusel. See lihtsustab oluliselt rakenduste ülesehitust ning vähendab infosüsteemi õppeperioodi uutele arendajatele. Docker konteinerite omavahelisi seoseid saab hallata OpenShift platvormil ning tänu sellele toimub rakenduste ehitamine erinevates keskkondades automaatselt.

## 3.8 Automaattestide käivitamine AS IS ja TO BE protsessis

AS IS protsessis on arendustiimidel kohustus katta uus kood vähemalt 70% ulatuses automaattestidega ning need käivitatakse automaatselt Jenkins nimelises tööriistas ühe osana töövoost rakenduse ehitamisel. Kui testide käivitamisel selgub, et kood käitub oodatust erinevalt, ebaõnnestub ka rakenduse ehitamise töövoog ning vead tuleb parandada. Kasutajaliidese automaatteste (Selenium) loovad testijad ning neid käivitatakse automaatselt mitu korda päevas. Lisaks testivad QA'd süsteemi ka manuaalselt.

TO BE protsessis suureneb automaattestide osatähtsus veelgi, sest eesmärk on muudatused tarnida kiiremini. See tähendab, et manuaalselt ei ole võimalik kõiki vigu piisavalt kiiresti ülesse leida ning kvaliteedi peavad tagama erinevad automaattestid. Lisaks UnitTestidele ja Selenium testidele on vaja kasutusele võtta automatiseeritud integratsioonitestid ning turvatestid, mida on võimalik automaatselt Jenkins ja OpenShift tööriistade abil Docker konteinerites olevate rakenduste peal käivitada.

## 3.9 Testkeskkonda üles seadmine AS IS ja TO BE protsessis

AS IS protsessis on rakenduste üles seadmine testkeskkonda Jenkins nimelise tööriista abil poolautomatiseeritud. See tähendab seda, et iga rakenduse tarbeks on loodud Jenkinsisse konfiguratsioon, kuidas rakendust ehitatakse, milliseid automaatteste käivitatakse ning kuidas ning millisesse serverisse failid kopeeritakse. Kogu töövoogu saab iga rakenduse puhul manuaalselt käivitada. Keskmiselt kulub iga töövoo sammu läbimisele 2-3 minutit, sest rakendustes oleva koodi hulk on suur. Lisaks eeldab rakendusest uue versiooni tarnimine ka veebiserveri taas käivitamist, mis tähendab kasutajatele katkestust ning ebamugavust. Rakendused võivad käituda erinevates keskkondades erinevalt, sest veebiserverid, mis rakendustega suhtlevad pole samad ning välised ressursid asuvad erinevates kohtades.

TO BE protsessis, kus kasutusele on võetud OpenShift platvorm saab ära kasutada olemasolevaid Jenkinsi konfiguratsioone. Lisaks sellele saab OpenShift platvormil kirjeldada ära rakenduse täiendavaid parameetreid nagu näiteks reeglid, millal suurendatakse süsteemi mastaapsust ja kui palju rakendus toimimiseks serveri ressurssi vajab. OpenShift platvormil on rakenduste ehitamine ja versioneerimine täielikult automatiseeritud ehk peale koodi lisamist Git koodihoidlasse, ehitatakse ja tarnitakse rakendus soovitud keskkonda automaatselt.

## 3.10 Toodangu keskkonda üles seadmine AS IS ja TO BE protsessis

AS IS protsessis toimub rakenduste toodangu keskkonda paigaldamine täielikult manuaalselt. See tähendab, et rakenduste ehitamine ja paigaldamine toimub käsurea abil ning kõigil teostajatel on olemas ka täielik ligipääs toodangu serveritele. Seetõttu on selline õigus väga väikesel hulgal arendajatel. Kõikide ettevõtte siseste kasutajatele mõeldud rakenduste puhul tähendab toodangu tarnimine ka kuni paari minutilist

katkestust töös. Iga rakenduse tarnimine toimub erineva loogika alusel, sõltuvalt sellest, millises tehnoloogias on rakendus realiseeritud ning milline on serveris olev veebiserver. Kuna rakenduste tarnimine on piisavalt keeruline protsess ja see põhjustab katkestuse kasutajate töös ning testimine teostatakse manuaalselt, toimub uute funktsionaalsuste toodangu keskkonda tarnimine keskmiselt kaks korda kuus.

TO BE protsessis toimub kõigi rakenduste toodangusse paigaldamine automaatselt peale kõigi vajalike testide positiivset läbimist või poolautomaatselt OpenShift tööriista kasutajaliidese kaudu. See tähendab, et toodangu serveritele ligipääsu pole vaja ning tarkvara tarnimist võib teostada ka äritellija talle sobival hetkel. Tänu mikroteenusarhitektuuri kasutamisele on muudatuste mõju kogu süsteemile oluliselt väiksem. OpenShift platvormi ja Docker konteinerite kasutusele võtmine võimaldab teha tarkvara tarne ilma süsteemi töö katkemiseta. Seetõttu on võimalik tarnida funktsionaalsust toodangu keskkonda oluliselt tihedamini kui seni.

## 3.11 Jälgimine AS IS ja TO BE protsessis

Jälgimine toimub AS IS protsessis valdavalt monitooringu skriptide abil ning automaatsete veateadete, mis saadetakse erinevatele osapooltele. Monitooringu skriptid tuleb luua rakendust arendades arendustiimi poolt ning igal rakendusel on need erinevad. Monitooringu skriptide poolt loodud väljundit jälgivad teenustoe spetsialistid ööpäevaringselt, kuid rakendusserveritele ligipääsu ning oskust spetsiifiliste rakenduste vigasid likvideerida neil üldiselt ei ole. Olgugi, et väga tihti piisab vaid rakenduse taas käivitamisest, et probleem laheneks ning süsteemi töö taastuks. Rakenduste logid asuvad rakendust majutavas serveris, mis tähendab, et probleemi lahendav arendaja vajab ligipääsu toodangu serverile.

TO BE protsessis on konteinerites olevate rakenduste jälgimine automatiseeritud. See tähendab, et OpenShift platvorm jälgib kõigi konteinerite toimivust ning vajadusel luuakse samast konteinerist uus versioon ja eelmine suletakse. Tegemist on automaatse süsteemi paranemisvõimega. Arendajatel on vaja sekkuda vaid juhul, kui rakenduse taas käivitamine ei anna soovitud tulemust. Lisaks sellele rakendatakse suurte koormuste puhul automaatselt mastaapsuse suurendamist ja hilisemat vähendamist rakenduste konteinerite lisamise või eemaldamisega, mis suurendab infosüsteemi töökindlust veelgi. Kõikide töötavate rakenduste logid kogutakse OpenShift platvormil automaatselt

eraldiseisvasse kogusse ning seetõttu kaob ära ka vajadus arendajatel pääseda ligi toodangu serverile.

# 4 Kokkuvõte

Magistritöö eesmärk oli leida DevOps arendusmetoodika ja põhimõtete realiseerimiseks vajaminevad infotehnoloogilised lahendused ning nende mõju tarkvaraarhitektuurile ning tööprotsessidele erinevates tarkvaraarenduse protsessi etappides. Kvalitatiivse uuringu olulisemad järeldused on, et tarkvara arendamisel tuleb lähtuda mikroteenusarhitektuurist ning rakenduste automaatseks tarnimiseks erinevatesse keskkondadesse on vajalik kasutusele võtta Docker konteinerid. Selleks, et muuta praegune *Continuous Integration* protsess *Continuous Deployment* protsessiks on vajalik kasutusele võtta privaatne pilvelahendus, kus on kasutusel OpenStack ja OpenShift serverihaldus- ja orkestratsioonitarkvara. See võimaldab mitmed manuaalsed protsessid automatiseerida ning seeläbi väheneb arenduste *time-to-market* aeg ning suureneb arendusprotsessi turvalisus, efektiivsus ja kvaliteet.

Magistritöös ei keskendutud TO BE protsessi võimalikele kitsaskohtadele ja realiseerimise keerukusele. Selles osas on võimalik teaduslikku tööd jätkata kvantitatiivse uuringu raames, kus vaadeldakse ja mõõdetakse reaalset mõju arendusprotsessi efektiivsusele ning rakenduste erinevatele jõudlusparameetritele peale soovitatud muudatuste realiseerimist.

# Kasutatud kirjandus


[1] Elisa Eesti AS, „Elisa koduleht,“ [Võrgumaterjal]. https://www.elisa.ee/et/elisast/organisatsioonist. [Kasutatud 26 April 2019].

[2] M. Virmani, „Understanding DevOps & bridging the gap from continuous integration to continuous delivery, *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, Pontevedra, Spain, 2015.

[3] R. Jabbari, N. bin Ali, K. Petersen ja B. Tanveer, „What is DevOps?: A Systematic Mapping Study on Definitions and Practices, *XP '16 Workshops Proceedings of the Scientific Workshop Proceedings of XP2016*, Edinburgh, Scotland, UK, 2016.

[4] V. Debroy, S. Miller ja L. Brimble, „Building lean continuous integration and delivery pipelines by applying DevOps principles: a case study at Varidesk, *ESEC/FSE 2018 Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA, 2018.

[5] D. Packer, „Continuous Integration vs. Continuous Delivery vs. Continuous Deployment,“ Plutora, 7 April 2019. [Võrgumaterjal]. https://www.plutora.com/blog/continuous-integration-continuous-delivery-continuous-deployment. [Kasutatud 26 April 2019].

[6] H. Kang, M. Le ja S. Tao, „Container and Microservice Driven Design for Cloud Infrastructure DevOps, *2016 IEEE International Conference on Cloud Engineering (IC2E)*, Berlin, Germany, 2016.

[7] P. Parmakson, „Mikroteenusarhitektuurist,“ Riigi Infosüsteemi Amet, August 2018. [Võrgumaterjal]. https://e-gov.github.io/TARA-Stat. [Kasutatud 7 Mai 2019].

[8] M. Shahin, M. Ali Babar ja L. Zhu, „The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives, *ESEM '16 Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real, Spain, 2016.

[9] M. Shahin, M. Ali Babar, M. Zahedi ja L. Zhu, „Beyond continuous delivery: an empirical investigation of continuous deployment challenges, *ESEM '17 Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Markham, Ontario, Canada, 2017.

[10] Y. Zhang, B. Vasilescu, H. Wang ja V. Filkov, „One size does not fit all: an empirical study of containerized continuous deployment workflows, *ESEC/FSE 2018 Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL, USA, 2018.

[11] Docker Inc., „Docker overview,“ 2019. [Võrgumaterjal]. https://docs.docker.com/engine/docker-overview/. [Kasutatud 8 Mai 2019].

[12] D. Jaramillo, D. V Nguyen ja R. Smart, „Leveraging microservices architecture by using Docker technology, *SoutheastCon 2016*, Norfolk, VA, USA, 2016.

[13] T. Rosado ja J. Bernardino, „An overview of openstack architecture, *IDEAS '14 Proceedings of the 18th International Database* , Porto, Portugal, 2014.

[14] Red Hat Inc., „Understanding OpenStack," Red Hat Inc., 2019. [Võrgumaterjal]. https://www.redhat.com/en/topics/openstack. [Kasutatud 9 Mai 2019].

[15] A. Khan, „Key Characteristics of a Container Orchestration Platform to Enable a Modern Application," *IEEE Cloud Computing,* pp. 42-48, Oktoober 2017.

[16] X. Cong, K. Rajamani ja W. Felter, „NBWGuard: Realizing Network QoS for Kubernetes, *Middleware '18 Proceedings of the 19th International*, Rennes, France, 2018.

[17] Kubernetes, „Kubernetes Concepts," 2019. [Võrgumaterjal]. https://kubernetes.io/docs/concepts/. [Kasutatud 9 Mai 2019].

[18] Elisa Oyj., „About Elisa," 2019. [Võrgumaterjal]. https://corporate.elisa.com/. [Kasutatud Mai 2019].

[19] A. Seppälä, *Case Study: OpenShift Architecture Evolution at Elisa OCG Helsinki 2018,* Helsinki: https://www.youtube.com/watch?v=JpTwSl_ZHfw, 2018.

[20] R. Kalbre, *Intervjuu Elisa tarkvaraarhitektiga.* [Intervjuu]. 10 Mai 2019.

[21] T. Lauringson, *Intervjuu 2 Elisa tarkvaraarhitektiga.* [Intervjuu]. 10 Mai 2019.

[22] R. Kullamaa, *Intervjuu Elisa juhtivarendajaga.* [Intervjuu]. 10 Mai 2019.