TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jelena Sarap 104671APM

AUTOMATISEERITUD KVALITEEDIKONTROLLI JA TAGASIPÖÖRDE VÕIMALUSEGA PIDEVJUURUTUSE PROTSESSI LOOMINE TELIA EESTI AS NÄITEL

Magistritöö

Juhendaja: Maili Markvardt
Magistrikraad

Tallinn 2021

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: Jelena Sarap

10.05.2021

Annotatsioon

Käesoleva lõputöö eesmärk on parendada arenduses KPI näitajad süsteemi töö efektiivsuse tõstmise kaudu. Selleks, et saavutada püstitatud eesmärki on vaja on analüüsida praegust pidevintegratsiooni (ingl k *Continuous Integration,* lühend *CI),* pidevvalmiduse (ingl k *Continuous Delivery,* lühend *CD),* pidevjuurutuse *(*ingl k *Continuous Deployment,* lühend *CD)* ning pidevmonitooringu (ingl k *Continuous Monitoring,* lühend *CM)* protsesse Telia mikroteenuse arhitektuuril *Reservation* rakenduse näitel. Tuvastada nendes protsessides puudujäägid, seejärel projekteerida ning realiseerida täiendatud ja optimaalsemaid protsesse.

Praegused pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsessid ei ole asjakohased, kuna hetkel puuduvad kasutajaliidese automaattestid. Riskantsete arendustööde puhul puudub võimalus neid teha kättesaadavaks lõppkasutajale järk-järgult, mis tähendab, et ainult osa kasutajatest näevad rakenduse uut versiooni. *Reservation* rakenduses praegu ei eksisteeri relevantset monitoorimise ja alarmide süsteemi.

Antud lõputöö raames autor realiseerib *Reservation* rakenduses kasutajaliidese automaattestid, kanaarväljalaske strateegia (ingl k *canary release*), lisab paindlikumat ja kindlamat monitooringut. Lisaks antud töös autor loob võimalust teostada automaatset tagasikeeramist rakenduse eelmisele versioonile juhul kui monitooringu vigade tase (ingl k *error rate*) on kriitiliselt kõrgem kui lubatud rakenduse *SLO* vigade eelarve (ingl k error budget*)*.

Rakenduse pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu praegust ja täiendatud protsessi, antud lõputöös, hindab autor rakenduse arendusprotsessi prognoositavate võtmenäitajate (ingl k *KPI*) abil *Reservation* rakenduse lõikes.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 59 leheküljel, 4 peatükki, 20 joonist, 7 tabelit.

Abstract

Implementing Continuous Deployment process by means of automated control and rollback in the example of Telia Eesti AS

The purpose of this thesis is to improve KPIs by increasing work effectiveness of the system. In order to achieve this purpose the author should analyze the current *Continuous Integration (CI)*, *Continuous Delivery (CD)*, *Continuous Deployment (CD)* and *Continuous Monitoring (CM)* processes in the *Reservation* application based on microservice architecture, to identify shortcomings in these processes and to design and implement an improved and optimal ones.

The current *CI*, *CD*, *CM* processes are not complete, as there are no user interface automatic tests. In the case of risky feature development, it is not possible now to release them step-by-step in such a way when feature is available only to a certain number of users. Also, the author sees a need to supplement monitoring and alarm system in this application to make decisions based on more accurate data.

As a result of this thesis, author updates the *Reservation* application with user interface automatic tests, adds capability to release application by means of canary release strategy. In addition, the author integrated advanced monitoring and designed a functionality that can perform an automatic rollback to the application version if the monitoring error rate is much higher than allowed within the application *SLO* error budget.

In this thesis the success of improved *CI*, *CD* and *CM* processes were evaluated by means of potential changes in Key Performance Indicators (*KPI*) of the *Reservation* application development process.

The thesis is in Estonian and contains 59 pages of text, 4 chapters, 20 figures, 7 tables.

Lühendite ja mõistete sõnastik

Alarm	Monitooringu element, mis informeerib jälgitava süsteemi võimalikust valest käitumisest
Alert	Monitooringu element, mis reageerib jälgitava rakenduse meetrika väärtuse muutmisele
Artifact	Protsessi tegevuse väljund
Backend	Rakenduse kasutajale peidetud osa, mis teostab äriloogikat ja andmete salvestamist
Bamboo	Pidevintegratsiooni ja -juurutuse server
Build	Paki ülesehituse protsess
CAPI	CRM andmebaasi nimetus
CI-FRONT	*Frontend* rakendus tagasihelistamiste ja esindustesse visiitide broneeringuteks
Covid-19	Haigus, mis on tingitud koroonaviirusest 2019-ndal aastal
Curl	Käsurea vahend andmete edastamiseks/pärimiseks, kasutades *URL* süntaksit
Docker	Programm, mida kasutatakse rakenduste väljalaskmise automatiseerimiseks ja juhtimiseks konteinerite toetavas keskkonnas
Error budget	Vigade taseme protsentuaalne väärtust: 100% - ***SLO***
Error rate	Vea tõenäosus, näidatakse tavaliselt protsentides
Feature haru	Haru, mis oli loodud *master* haru alusel ja kus asub uus koodimuudatust enne *merge* master haruga
Frontend	Rakenduse kasutajale nähtav osa, mis töötab kliendi pool
Grafana	Rakenduste meetrikatel põhinev veebirakendus analüütika ja visualiseerimise jaoks
Graylog	Logide agregaator
Image	Mall koos instruktsioonidega konteineri ehitamiseks selleks, et käivitada *Docker* platvormil
Jira tellimus	Arendustöö tellimus, mis on vormistatud *Jira* rakenduses
Juhtimiskeskus	Osakond, mis juhib kõiki tegevusi
Kanaarjuurutus	(ingl k *Canary deployment*) Rakenduse juurutuse tehnika, mis paigaldab rakenduse uut versiooni piiratud eksemplarides

Kanaarväljalase	(ingl k *Canary release*) Rakenduse väljalaskmise tehnika, mis teeb rakenduse uut versiooni kättesaadavaks ainult osale kasutajatest
Klaster	Sõlm-masinate kogum, *Docker* konteiner rakenduste käivitamiseks *Kubernetes* platvormil
Koormusejaotur	(ingl k *Load Balancer*) Seade, mis käitub nagu *reverse proxy-server*, jagades võrguliiklust ja koormust serverite vahel optimaalsemal viisil
KPI	(ingl k *Key Performance Indicators*) efektiivsuse võtmemõõdikud
Kubernetes	Platvorm rakenduste konteinerite orkestreerimiseks ja skaleerimiseks
Küllastumine	(ingl k Saturation) näitab kui palju ressursse oma maksimumist kasutab rakendus praegu
Master haru	Haru, kus asub primaarne ja stabiilne rakenduse koodi seis
Merge	Koodimuudatuse integratsioon koodi seisuga *master* harus
Mikroteenus	Teenus, mis teostab oma ülesandeid, töötab omas protsessis ja kommunikeerub teistega kasutades HTTP protokolle
Namespace	*Kuberetes* klastri loogiline jaotus selleks, et erinevad kasutajad saaksid kasutada samat klastrit, olles ühendatud sama loogikaga
Nginx	*HTTP* ja *reverse proxy-server*, mida saab kasutada ka rakenduse koormuse jaoturiks
Pidevintegratsioon	**C**ontinuous **I**ntegration ***(CI)*** - rakenduse pideva integratsiooni protsess
Pidevvalmidus	**Continuous D**elivery ***(CD)*** - rakenduse pideva valmiduse väljalaskmiseks protsess (inimese sekkumisega enne väljalaskmist toodangu keskkonda)
Pidevjuurutus	**Continuous D**eployment ***(CD)*** – rakenduse pideva juurutuse protsess (ilma inimese sekkumiseta)
Pidevmonitooring	**Continuous M**onitoring ***(CM)*** – rakenduse pideva monitoorimise protsess
Pipeline	Andmetega opereeriv elementide jada, kus eelmise elemendi väljund on järgmise sisend
Plumbr	Eesti tarkvara, mis teeb funktsionaalset monitooringut kasutaja brauseri ja serveri poolt
Pod	Kõige väiksem element, mida saab panna püsti *Kubernetes* platvormile. Selle peal töötavad rakenduste konteinerid
Production	Rakenduse toodangu keskkond, mis on kättesaadav lõppklientidele

Prometheus	Monitooringu ja teadete (ingl k *alert*) süsteem ja mälusisene dimensiooniline aegridade andmebaas (ingl k *time series database*)
Reservation rakendus	Esinduse aja broneerimise ja tagasihelistamise rakendus
Reverse proxy-server	*Proxy-server*, mis edastab kasutaja päringut teistele serveritele, kuigi kasutaja jaoks see näeb välja nagu kõik ressursid asuvad antud reverse *proxy-serveril*
Quality Manager	Isik, kelle vastutuses on rakenduste kvaliteedi kontrolli organiseerimine
Shell Script	Programm, mida käivitatakse *Unixi* töökäskude interpreteerijas
SLA	(ingl k *Service-level Aggreements*) teenusetaseme kokkulepped
SLI	(ingl k *Service-level Indicators*) teenusetaseme indikaatorid
SLO	(ingl k *Service-level Objectives*) teenusetaseme eesmärgid
Slack Messenger	Korporatiivne kommunikatsiooni platvorm
Splunk	*USA* firma, monitooringu tarkvara arendaja
Stackholder	Osapool, kes on huvitatud funktsionaalsuse teostamises ja kvaliteedis
Stage	Rakenduse toodangu keskkonnale eelnev keskkond, kus tihti teostatakse lõplikku integratsiooni testimist toodangu konfiguratsiooniga
Testcafe	*JavaScripti library* kasutajaliidese automaattestide arendamiseks
TypeScript	Programmeerimiskeel, mis laiendab *JavaScripti* funktsionaalsust ja omab tüübikindlat süntaksit
TWEB	Avaliku veebi rakendus, kus näidatakse esinduste kontaktinfot
Väljalase	(ingl k *release*) rakenduse uue versiooni avalikustamine
Zabbix	Platvorm võrgu, serverite, teenuste, pilveressursside ja ärinäitajate monitooringu jaoks

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Antud lõputöö teema on „Automatiseeritud kvaliteedikontrolli ja tagasipöörde võimalusega pidevjuurutuse protsessi loomine Telia Eesti AS näitel“. Autor analüüsib pidevintegratsiooni (ingl k. *Continuous Integration*, lühend *CI*), pidevvalmiduse (ingl k. *Continuous Delivery*, lühend *CD*), pidevjuurutuse (ingl k. *Continuous Deployment*, lühend *CD*) ja pidevmonitooringu (ingl k. *Continuous Monitoring*, lühend *CM*) protsesse, otsib mustrit efektiivsete protsesside jaoks ning rakendab neid *Reservation* rakenduse näitel. Lõputöös vaadeldud teemad olid uuritud, projekteeritud ning realiseeritud autorina olles *Quality Manager* rollis.

1.1 Taust ja problemaatika

Telia Eesti AS on suur telekommunikatsiooni ettevõte, mis kuulub rahvusvahelisse ettevõttesse - Telia Company AB. Praegu Telia Eesti AS ettevõttes töötab rohkem kui 1600 inimest. Ettevõte kasutab nii väliste, kui ka siseklientide otstarbeks suurt hulka rakendusi. Olles üks telekommunikatsiooni teenuste turu liidritest, teeb ettevõte parimat, et tema teenuste kvaliteet ja kasutajakogemus oleksid kõrgtasemel. Seega sõltub väga palju IT teenuste arendusprotsessi efektiivsusest. Kuigi praegune arendusprotsess Telias on juba väga arenenud ja rakendab palju maailma parimaid praktikaid, eksisteerivad kohad testimises, rakenduste versioonide väljalaskmises ning monitooringus, mida saab muuta veel efektiivsemaks. Seega antud lõputöö eesmärk on muuta arenduse KPI väärtused paremaks ühe rakenduse alusel nimega *Reservation*. Arenduse *KPI* väärtused on oluline osa arendusvaldkonna aasta hindamiseks, seega nende paremaks muutmine on väga aktuaalne. Püstitatud hüpotees seisnes selles, et antud eesmärki võib saavutada pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside täiendamise kaudu. *Reservation* rakendus oli üleantud CRM valdkonna arendustiimile, kus töötab antud lõputöö autor. Antud rakenduse kaudu Telia Eesti kliendid ja teenindajad klientide nimelt saavad tellida tagasihelistamisi ja reserveerida visiidi aega Telia esindustesse, töödelda reserveeringuid ning muuta informatsiooni Telia esinduste kohta. Antud funktsionaalsus muutus olulisemaks *Covid-19* ajal, kui inimesed tahtsid vähem kokku puutuda omavahel ning kõnede osakaal kasvas. *Reservation* rakenduses puudusid teatud

aspektid pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessidest. Lisaks, rakendusele oli vaja määrata teenusetaseme eesmärke (ingl k *SLO, Service-level Objectives*) ning teenusetaseme indikaatoreid (ingl k *SLI, Service-level Indicators*) selleks, et nende alusel tagada kontroll kvaliteedis [1].

Puudused *Reservation* rakenduse praeguses implementatsioonis:

- puuduvad kasutajaliidese integratsiooni automaattestid;
- rakenduse versiooni väljalaske protsessis puudub võimalus teostada kanaarjuurutust;
- monitooringu protsessis jääb suur hulk võimalikke meetrikaid tähelepanuta;
- töövälisel ajal kriitilised vead jäävad märkamata;
- puudub automaatse rakenduse versiooni tagasikeeramise võimalus juhul kui vigade tase on palju kõrgem kui kriitiline määr.

1.2 Töö ülesanded

Antud lõputöös planeerin:

- analüüsida pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu praktikate teoreetilist materjali;
- läbi viia küsitluse ning selle alusel luua protsesside mustrit, mis sobib Telia CRM valdkonna rakendustele;
- teostada analüüs *Reservation* rakenduse praeguste pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesside kohta ning püstitada täiendavad nõudmised uuendatud protsessidele;
- võrrelda praeguseid ja uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesse
- realiseerida uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu lahendust;

- esitada järeldusi *KPI* (***K**ey **P**erformace **I**ndicators*) potentsiaalsete tulemuste alusel;

- esitada ettepanekuid edasiste tegevuste kohta.

Antud lõputöö uuring ja realisatsioon on kooskõlastatud autori otsese juhiga ja tema poolt heaks kiidetud.

1.3 Metoodika

Autor on arutanud koos Telia Eesti erinevate valdkondade arhitektidega, arendajatega, administraatoritega ja äritellijatega, mis oleksid arendusprotsessi edasised arengukohad. Lisaks autor viis läbi küsitluse pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu kohta, mille tulemustest valis välja aspektid edasiseks realisatsiooniks. Osa edasise realisatsiooni töödest on planeeritud jätkutöödena. Selleks, et kirjeldada praegust arenduse ja monitooringu protsessi olid loodud *BPMN* diagrammid, ehk protsessi diagrammid. Kõik diagrammid olid teostatud *Draw.io* programmi abil.

Realisatsioon oli loodud, kasutades programmeerimiskeeli: *Testcafe*, *TypeScript*, *bash*, *cURL*, *Java*. Teostuse käigus olid loodud *Kubernetes* objektide ja *Nginx*-i konfiguratsioonid, kirjutatud kasutajaliidese autotestid, loodud *bash* skriptid ning *Javas* konfigureeritud *Prometheus* meetrikad. Realisatsioon on seletatud vastavates peatükkides koodinäidete ja jooniste abil. Plokist „Lisad“ võib leida antud realisatsiooni detailsemad lõigud. Realisatsiooni failid on lisatud antud töö manuses (tööfailid).

Järeldused olid tehtud realisatsiooni tulemuste ning *KPI* väärtuste alusel (tulevane potentsiaalne muudatus).

1.4 Töö ülesehitus

Töö on jaotatud neljaks loogiliseks osaks.

Esimeses osas autor kirjeldab erinevaid praktikaid kuidas ehitatakse üles pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesse.

Lisaks autor viib läbi küsitluse oma kolleegide seas, kuhu kuuluvad arendajad, arhitektid, administraatorid ja valdkonna juhataja. Näitab kuidas näevad välja praegused protsessid

ning nende uuendatud variant. Peatüki lõpus autor toob välja vana ja uuendatud protsesside erinevused.

Teises peatükis autor kirjeldab uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesside realisatsiooni, tuues välja ka alternatiivseid tehnilisi lahendusi. Sinna kuuluvad sellised alamosad nagu automaattestimine, kanaarjuurutus, monitooring ning automaatne tagasipööre rakenduse eelmisele versioonile.

Kolmandas peatükis kirjeldab autor järeldusi tehtud töö põhjal.

Neljandas peatükis autor kirjeldab edasisi plaane.

2 Pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessid

Antud peatüki eesmärk on tutvustada teoreetilist materjali pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside teemadel, kirjeldada praeguseid protsesse, välja selgitada nende protsesside probleemsed kohad ning välja kujundada nõudmised uutele protsessidele.

2.1 Pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu praktikad

Pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessid kujutavad endast kombineeritud praktikaid, mis kohustavad kasutama automatiseerimist rakenduse paki ehituse, testimise, juurutamise ning monitooringu faasidel [2].

Pidevintegratsiooni (ingl k. *Continues Integration*) põhifookuseks on automaattestide käivitamine pärast igat koodimuudatuse *merge* operatsiooni *master* haruga. Tihti automaatteste käivitatakse ka vastu *feature* haru. Pidevintegratsiooni automaattestide alla võivad kuuluda nii rakenduse ühikutestid (*Unit tests*) [3] kui ka kasutajaliidese autotestid (*Acceptance and Regression tests*) [4][5]. Selline praktika tagab kiiremat tagasiside tsüklit uue koodimuudatuse sobivuse kohta. Lisaks, see annab võimalust vähendada testijate osakonna koormust ja suunata nende tähelepanu, näiteks, uurimistestimisele (*Exploratory testing*)[1] või ümber kvalifitseerida manuaaltestijad analüüsi, arendamise ja administreerimise poole, rakendades *T-shape skills* mudelit [6].

Pidevintegratsiooni protsessi võivad kuuluda erinevad koodi kontrollid ja testide liigid. Juhul kui tegu on *Java* ja *JavaScripti* rakendusega, siis tegevuste ja tehnoloogiate nimekiri võib välja näha järgmiselt:

[1] https://ieeexplore.ieee.org/abstract/document/1541817

- koodi staatiline analüüs, kasutades vahendeid nagu ESLint[1], Prettier[2] ja teisi;
- mockitud ühiktestid rakenduse *frontend* ja *backend* osades koos kaetuse analüüsiga nagu *JUnit*[3] + *Mockito*[4], *Jest*[5], *istanbul*[6] ja *Jacoco*[7];
- integratsioon *backend* testid (*Service integration tests*), kasutades sama *JUnit*;
- *tap-comparison* testid (testid dubleerivad toodangu keskkonna päringuid ja kontrollivad rakenduse uues versioonis, mis on juurutatud, kuid mitte veel reliisitud) [7];
- kasutajaliidese integratsiooni testid (*Acceptance tests*), kasutades *Testcafe*[8], *Cypress*[9], *Selenium*[10], *Watir*[11] ja teisi. Kasutajaliidese integratsiooni testid võivad olla ka pidevvalmiduse ja -juurutuse protsesside osa;
- kasutajaliidese regressiooni integratsiooni testid (*Regression tests*), kasutades *Testcafe*, *Cypress*, *Selenium*, *Watir Splash* ja teisi. Kasutajaliidese regressiooni integratsiooni testid võivad olla ka pidevvalmiduse ja -juurutuse protsesside osa;
- koormustestid (*Performance tests*) [8]. Antud testid võivad olla ka pidevvalmiduse ja -juurutuse protsesside osa;
- rakenduse turvalisuse testid (*Penetration tests*) [9]. Antud testid võivad olla ka pidevvalmiduse ja -juurutuse protsesside osa.

[1] https://eslint.org/

[2] https://prettier.io/docs/en/index.html

[3] https://junit.org/

[4] https://site.mockito.org/

[5] https://jestjs.io/

[6] https://istanbul.js.org/

[7] https://www.eclemma.org/jacoco/

[8] https://testcafe.io/

[9] https://www.cypress.io/

[10] https://www.selenium.dev/

[11] http://watir.com/

Pärast seda kui koodimuudatus läheb *feature* või *master* harudesse koodihoidlas, pidevintegratsiooni server käivitab osa või kõike nendest kontrollidest. Juhul kui mingi protsess lõpetab veaga, pidevintegratsiooni server ei saa üles ehitada rakenduse *artifact-e* järgmisteks faasideks ning antud haru koodimuudatuse autorid saavad sellest teadet.

Pidevvalmidus (ingl k *Continuous Delivery*) on pidevintegratsiooni protsessi laiend. Antud protsess, kasutades *artifact-e,* mis olid saadud pidevintegratsiooni protsessi käigus, teeb juurutust *(*ingl k *deploy)* järgmistesse keskkondadesse: arendus -ja testkeskkonnad (*Dev ja Stage*). Seega pidevintegratsiooni protsess jätkub, valmidusega järgmise sammuna teostada automaatne väljalase (ingl k *release*). Pidevvalmiduse protsessi suhtes on vaja pöörata tähelepanu sellele, et see ei tee automaatset juurutust toodangu keskkonda (*Production*), vaid ootab inimese sekkumist kas või minimaalselt ühe nupu vajutamise vormis.

Antud etapil tihti kasutatakse järgmisi tegevusi:

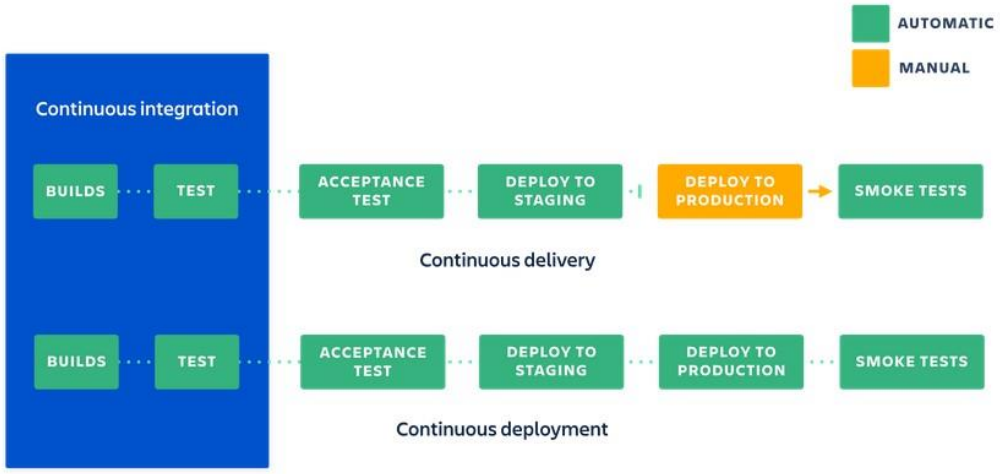
- kasutajaliidese regressiooni integratsiooni teste (*Regression tests*);
- koormuse teste (*Performace tests*);
- rakenduse turvalisuse teste (*Penetration tests*).

Juhul, kui need testimise liigid lõpetasid oma töö ilma vigadeta, rakenduse versioonide väljalaske konveieri (ingl k *Pipeline*) triger võib liigutada väljalaske kandidaati järgmisele etapile, nimelt – toodangu keskkonnaks valmis (*Ready for production*). Pärast seda kui koodimuudatusega pakk pannakse üles toodangu keskkonda, võivad käivituda ka kasutajaliidese integratsiooni testid, mis hõlmavad kõige kriitilisemat ärifunktsionaalsust (*Production Smoke tests*) [10].

Pidevjuurutus *(ingl k Continuous Deployment)* astub ühe sammu edasi, võrreldes pidevvalmiduse protsessiga. See loob sellist protsessi, kus testimise ja rakenduse versioonide väljalaskmise konveieril (ingl k *Pipeline)* ei jää enam kohta manuaalseteks tegevusteks nagu väidavad oma raamatus J. Humble ja D. Farley [12]. See tähendab, et juhul kui kõik automaatsed kontrollid igal faasil lähevad läbi, uus versioon rakenduse koodist läheb automaatselt klientidele.

Samamoodi nagu pidevvalmiduse puhul, pärast seda kui koodimuudatusega pakk pannakse üles toodangu keskkonda, võivad käivituda ka kasutajaliidese integratsiooni testid, mis hõlmavad kõige kriitilisemat ärifunktsionaalsust (*Production Smoke tests*).

Visuaalselt pidevintegratsiooni, -valmiduse ja -juurutuse protsesse võib kuvada järgmise skeemina [Joonis 1]



Joonis 1. Pidevintegratsiooni, -valmiduse, -juurutuse skeem [11].

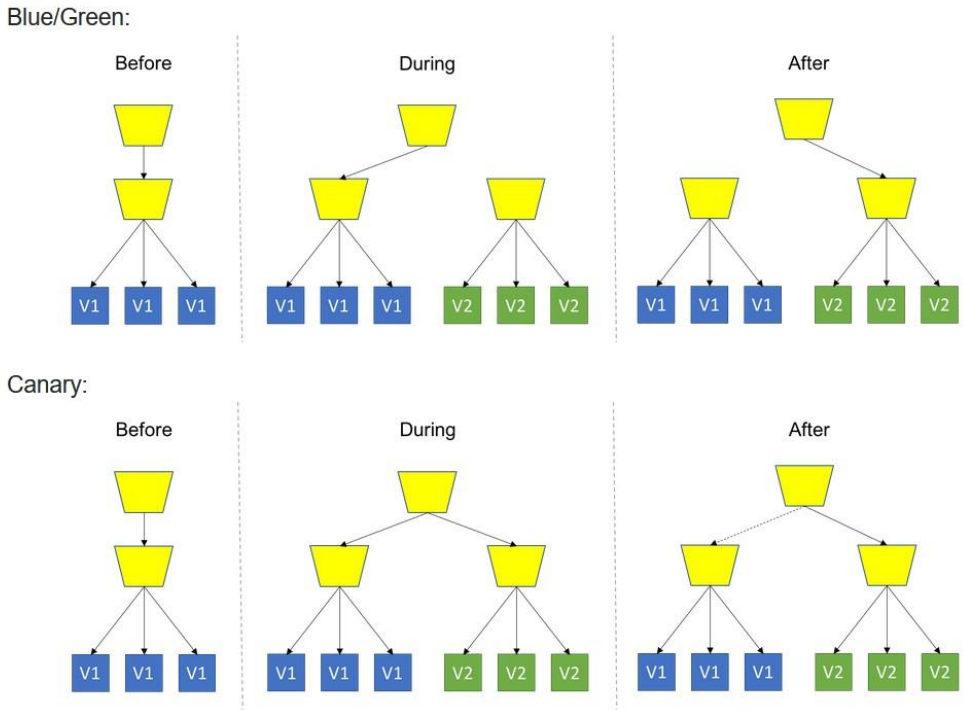
Pidevjuurutuse raames autor käsitleks ka väljalaskmise strateegiaid. Praegu kõige põhilisemad nendest on [13]:

- katkestusega rakenduse uue versiooni väljalaskmine;
- katkestuseta rakenduse uue versiooni väljalaskmine sinirohelise väljalaske strateegia abil (ingl k *blue-green strategy*);
- rakenduse uue versiooni väljalaskmine kanaarjuurutamise strateegiaga (ingl k *canary release*);
- tume käivitamine (ingl k *Dark Launch*)
- rakenduse uue versiooni väljalaskmine kasutades lippe (ingl k *flag, A/B testing).*

Katkestusega väljalaskmise raames, rakenduse uue versiooni paigaldamises toimub lühiajaline rakenduse töö katkestus. Sellisel juhul tavaliselt tehakse paigaldusi öisel ajal.

Katkestuseta rakenduse uue versiooni väljalaskmine eeldab, et võrguliiklust saab ümber suunata kui tulevad püsti rakenduse eksemplarid koos uue versiooniga. Kanaarväljalaskmise strateegia eeldab, et rakenduse teatud hulk eksemplare uueneb ning sinna suunatakse eelnevalt määratud hulk võrguliiklusest. Rakenduse versioonide väljalaskmine kasutades lippe võib toimida koos nii sinirohelise kui ka kanaarväljalaskmise strateegiaga ja eeldab, et rakenduse konkreetset koodi näeb ainult teatud osa klientidest. Kanaarjuurutuse ja lippude strateegiate eesmärk on tagada turvalist testimist toodangu keskkonnas ning minimiseerida riske, mis võivad ilmuda rakenduse uue versiooni kasutusele võtmisel. Tume käivitamine toimib sarnaselt kanaarjuurutusega, aga selle erinevusega, et uue koodi versiooni vastused ei jõua klientideni.

Allolev joonisel [Joonis 2] näitab võrdluseks kuidas teostatakse rakenduse versioonide sinirohelise ja kanaarväljalaske strateegiaid:



Joonis 2. Siniroheline väljalase vs kanaarjuurutuse strateegia [14].

Pidevmonitooring (ingl k *Continuous Monitoring*) mõiste ei ole nii levinud, võrreldes pidevintegratsiooni, -valmiduse, -juurutuse mõistetega, kuid see käib väga tihedalt paaris koos pidevjuurutusega, tagades viimasele töökindlamat protsessi. Pidevmonitooringut võib käsitleda ka pidevjuurutuse protsessi lahutamatuks osaks.

Alloleval joonisel [Joonis 3] on näha kus asub pidevmonitooringu plokk pidevjuurutuse protsessis:

Joonis 3. Pidevmonitooring pidevjuurutuse protsessis [15].

Pidevmonitooring koosneb enamasti järgmistest tegevustest [16]:

- logide korjamine ja jälgimine masinal, kus jälgitav rakendus töötab;
- sõltuvate rakenduste logide agregeerimine ühes monitooringu süsteemis ja seal nende vaheliste sõltuvuste jälgimine (ingl k *Tracing*) [17];
- logide abil rakenduse meetrikate korjamine
- meetrikate põhjal raportite visualiseerimine
- monitooringu süsteemides infoteadete ja veateadete sätestamine;
- monitooringu süsteemides rakenduste enesediagnostika ja eneseparandamine.

R. Ewaschuk Google SRE raamatus [18] toob välja neli kuldset signaali, mida hea monitooringu visualiseerimiskiht peab rakenduses jälgima . Need signaalid on järgmised:

- Vead (ingl k *Errors*)
- Võrguliiklus (ingl k *Traffic*) – kui palju HTTP päringuid on mingil ajahetkel
- Latentsus (ingl k *Latency*) – viivitus vastuste töötlemises

- Küllastus/küllastumine (ingl k *Saturation*) – kui palju ressursse oma maksimumist kasutab rakendus praegu. Küllastumise alusel võib teha ennetusi.

Need on põhiaspektid, mida on vaja tagada rakenduse monitooringu süsteemis ja mis aitaksid hoida rakenduse seisu kontrolli all. Lisaks, nendele neljale signaalile, on olemas ka ärifunktsionaalsuse meetrikad, mille alusel võib teha statistikat rakenduse kasutatavuse kohta.

Pidevjuurutuse protsessist tingitud tihedam rakenduse muutmine vajab veel intensiivsemat monitooringu protsessi, mis ei saa enam olla manuaalne. Manuaalne monitooring on üks nõrkadest lülidest, mis võib aeglustada pidevjuurutuse protsessi. Seega on väga oluline, et monitooring oleks piisavalt paindlik selleks, et dünaamiliselt kohaneda funktsionaalsuse kasvuga ja erinevatel tasemetel reageerida rakenduse probleemidele.

Pidevmonitooringu protsessiga on tihedalt seotud sellised mõisted nagu *SLA* (***S**ervice-**l**evel **A**greement*), *SLO* (***S**ervice-**l**evel **O**bjectives*) ja *SLI* (***S**ervice-**l**evel **I**ndicator*) ja vigade eelarve (ingl k *error budget)* millest detailselt räägivad C. Jones, J. Wilkes, N. Murphy ja C. Smith Google SRE raamatu neljas peatükis [19].

SLA – teenusetaseme kokkulepe teenuseosutaja ja kliendi vahel mõõdetavate näitajate kohta, nagu näiteks, reageerimisaeg ja administraatori kättesaadavus. Selle kokkulepe mitte täitmine tavaliselt toob kaasa rahalisi karistusi.

SLO – kokkulepe teenusetaseme eesmärkidest teenuseosutaja ja välis – või sisekliendi vahel, kus on pandud kirja konkreetsed tehnilised ootused rakendusele, nagu näiteks, rakenduse töö kestvus (ingl k *uptime*) ja reageerimisaeg (ingl k *response time)* kvartalis. Antud kokkulepe kindlasti kooskõlastatakse IT osakonnaga.

SLI – teenusetaseme indikaatorid, konkreetsed mõõdikud, mida hakatakse jälgima, selleks, et olla kursis kas lubatud *SLO* ja *SLA* näitajad on tagatud või mitte.

Vigade eelarve – vigade arv protsentides, mida rakendus võib lubada teatud ajaperioodil. Seda võib arvutada antud valemi järgi 100% - *SLO*%.

Pidevintegratsioon, -valmidus ja -juurutus võivad hõlmata palju tegevusi ning autori arvamusel kõige olulisemad neist olid kirjeldatud antud peatükis. Selleks, et paremini aru

saada missugused kirjeldatud praktikatest tunduvad perspektiivsetena ja efektiivsetena Telia IT osakonna kollektiivile, otsustas autor läbi viia küsitluse pidevintegratsiooni, -valmiduse, -juurutamise ning -monitooringu erinevate tegevuste kohta. Küsitlusest detailsemalt räägib järgmine 2.2 peatükk.

Selleks, et teha kokkuvõtet teoreetilisest materialist, mis on pühendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessidele, autor pakub pöörata tähelepanu sellele, et kõikide nende protsesside eesmärk seisneb selles, et tagada kiiret arendus -ja tagasisidetsüklit, mille raames:

- uued arendused jõuavad võimalikult kiiresti lõppkasutajateni (KPI: Arenduse aeg (*Dev time*)↓, KPI: Aeg uue funktsionaalsuse väljalaskmiseni (*Time to market*)↓),
- tagatakse maksimaalset võimalikku kvaliteeti (KPI: Kriitiliste ja oluliste vigade arv toodangu keskkonnas (*Critical and Major Live Bugs Count*)↓, KPI: Toodangu keskkonna vigade keskmine lahendamise aeg (*Live bugs resolution average time*)↓).

On vaja leida selliseid protsesse ja tegevusi, mis toetavad maksimaalselt antud eesmärki.

2.2 Pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesside küsitlus

Autor on teostanud küsitluse, kus ta uurib mida arendajad, arhitektid, administraatorid ja juhid peavad pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessides vajalikuks ning millistest tegevustest oleks mõistlik loobuda.

Küsitluse võib jagada neljaks osaks:

- Arendus ja testimine
- Rakenduse versiooni väljalase
- Monitooring
- Ettepanekud olemasolevate pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside kohta

Kokku küsimusi oli 31. Vastuste variandid määravad vastaja nõusolekut küsimuse väitega. Selle jaoks, et määrata tendentse, oli välja arvutatud vastuste kaalutud keskmine. Iga vastuse variandile on omistatud punktide arv 5 punkti süsteemi järgi. Vastuste variandid olid järgmised:

- Ei ole üldse nõus / Ei tea midagi – 1 punkt
- Pigem ei ole nõus / Pigem ei tea – 2 punkti
- Ei oska öelda / Ei oska öelda – 3 punkti
- Pigem nõustun / Pigem tean – 4 punkti
- Olen täiesti nõus / Tean – 5 punkti

2.3 Küsitluse tulemused

Küsitluses oli saadud 12'lt vastajalt vastused. Nende hulgas oli 5 arendajat, 5 arhitekti, 1 administraator, 1 osakonna juhataja.

Esimesed küsimused olid pühendatud sellele, kas inimesed tunnevad pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu teemasid. Hiljem sama küsimus oli esitatud kanaarjuurutuse kohta. Allolev tabel [Tabel 1] näitab kuidas protsentuaalselt jaotusid vastused:

Tabel 1. Pidevintegratsiooni, -valmiduse, -juurutuse, -monitooringu ja kanaarjuurutuse mõistete tundmine.

Küsimus	**KK**
1.Kas te teate, mida tähendab pidevintegratsioon (*CI*)?	4,42
2.Kas te teate, mida tähendab pidevvalmidus (*CD*)?	4,25
3.Kas te teate, mida tähendab pidevjuurutus (*CD*)?	4,42
4.Kas te teate, mida tähendab pidevmonitooring (*CM*)?	3,83
5.Kas te teate, mida tähendab kanaarjuurutus?	4,08

CI – pidevintegratsioon, *CD* – pidevvalmidus/-juurutus, *CM* – pidevmonitooring, KK – kaalutud keskmine

Saadud tulemustest võib järeldada, et enamik vastajatest teavad mida tähendab pidevintegratsioon, -valmidus, -juurutus, -monitooring ning kanaarjuurutus. Kuigi

paistab silma, et mõiste pidevmonitooring ei ole nii levinud ja tõe poolest seda tihti käsitletakse pidevjuurutuse osana. Aga antud töös autor pakub käsitleda antud protsessi eraldi, kuna pidevjuurutuse strateegiat võib realiseerida ka ilma pidevmonitooringu protsessita. Lisaks, küsitluse tulemused näitasid, et vahest on raske eristada pidevvalmidust ja pidevjuurutust.

Järgmised küsimused antud lõputöö küsitluses puudutasid vastajate nõusolekut pakutud väidete kohta ning „Arendus ja testimine“ ploki vastustega võib tutvuda allolevas tabelis [Tabel 2]:

Tabel 2. Arendus ja testimine.

Küsimus	**KK**
6.Koodi staatiline analüüs tõstab koodi ja rakenduse kvaliteeti ning on seetõttu väärt kuluvat aega	3,92
7.Ühik- ja integratsiooni testid tõstavad *CI* protsessis rakenduse kvaliteeti ning on seetõttu väärt kuluvat aega	4,75
8.Tap-comparison testid *CI/CD* protsessis (testid dubleerivad toodangu päringuid ja kontrollivad uues versioonis, mis on juurutatud, kuid mitte veel väljalastud) tõstavad rakenduse kvaliteeti ning on seetõttu väärt kuluvat aega	3,67
9.*Mock*-itud kasutajaliidese testid *CI* protsessis tõstavad rakenduse kvaliteeti ning on seetõttu väärt kuluvat aega	4,08
10.Kasutajaliidese integratsiooni testid rakenduse *CI* protsessis tõstavad rakenduse testimise kvaliteeti ning on seetõttu väärt kuluvat aega	4,42
11.Manuaalse testimise faas *feature* haru arendusprotsessis tagab paremat rakenduse kvaliteeti ning on seetõttu väärt kuluvat aega	4,33
12.Turvalisuse testid (teenuste ligipääsu (has_permission) volituste testimine) tagavad rakenduse paremat kvaliteeti ning on seetõttu väärt kuluvat aega	4,17
13.Sõltuvate rakenduste lokaalne arendus/automaatne testimine ühes komplektis (näiteks: *TWEB*, *CI-FRONT*, *Reservation*, *CAPI*) tagaks paremat testimist ja mugavat arendust ning seetõttu selle võimaluse realisatsioon on väärt kuluvat aega	2,83
14.Automaattestide (ühiktestid, integratsiooni testid, kasutajaliidese integratsiooni autotestid) käivitamine igas harus aitab avastada rakenduse vigu varasemal faasil	4,67
15.Kasutajaliidese autotestid (*Production* S*moke tests*) toodangu keskkonnas tõstaksid rakenduse testimise veel paremale tasemele ning on seetõttu väärt kuluvat aega.	3,58

16.Rakenduse koormustestid testkeskkonnas (S*tage*) tõstaksid rakenduse testimise veel paremale tasemele ning on seetõttu väärt kuluvat aega.	3,83

CI – pidevintegratsioon, *CD* – pidevvalmidus/juurutus, KK – kaalutud keskmine

Saadud vastustest on näha, et vastajad kõige rohkem toetavad ühik -ja integratsiooni teste (*backend*) ja on nõus nendesse investeerima oma aega, sest arvavad, et see tagab rakenduse paremat kvaliteeti (kaalutud keskmine - 4,75 ja 4,67). Natuke vähema aga ikkagi päris kõrge hinnangu said kasutajaliidese integratsiooni autotestid (kaalutud keskmine - 4,43). Vastused näitasid, et vastajad on arvamusel, et *feature* harude manuaalne testimine parandab rakenduse kvaliteeti ning on kuluvat aega väärt (kaalutud keskmine – 4,33). Autor võib sellest järeldada, et manuaalse testimise faas peab jääma pidevintegratsiooni protsessi. Kõrge hinnangu said ka rakenduse tasemel automaatsed turvatestid ning *mock*-itud kasutajaliidese autotestid (kaalutud keskmine – 4,17 ja 4,08). Need autotestide liigid on praegu aktiivselt kasutusel ka olemasolevates rakendustes. Järgmised autotestide liigid ei ole praegu kasutusel, kuid lähtudes küsitluse tulemustest nende kontseptsioon meeldib vastajatele:

- automaatsed koormustestid (kaalutud keskmine – 3,83);
- *tap-comparison* autotestid (kaalutud keskmine – 3,67);
- automaatsed testid toodangu keskkonnas (*Production Smoke tests*) (kaalutud keskmine – 3,58).

Koodi staatiline analüüs sai ka päris kõrge hinnangu (kaalutud keskmine – 3,92). Antud praktika on aktiivselt kasutusel paljudel rakendustel. Tasub märkida, et see hinnang on madalam kui rakenduste olemasolevatel autotestide liikidel. Autor võib järeldada, et antud praktika kasutamisega on osadel vastajatel seotud pigem negatiivsem kogemus.

Ootamatu tulemuse andsid küsimused mikroteenuste autotestimisest ning arendusest ühes komplektis (kaalutud keskmine – 2,83). Võib-olla andis mõju ideoloogiline arusaam sellest, et mikroteenused olid loodud selleks, et neid oleks võimalik arendada ja testida eraldi. Igal juhul küsitlusest saadud hinnang vajab arutelu ja lähemat uurimist.

Ploki prioriteedid on järgmised:

- Ühik-ja integratsiooni autotestid *backend* jaoks

- Kasutajaliidese integratsiooni autotestid
- *Mock*-itud autotestid *frontend* jaoks
- Automaatsed turvatestid
- Koodi staatiline analüüs
- Automaatsed koormustestid
- *Tap-comparison* testid
- Automaatsed testid toodangu keskkonnas (*Production Smoke tests*)

Järgmine tabel [Tabel 3] illustreerib vastuseid, mis oli saadud plokis „Rakenduse versiooni väljalase“:

Tabel 3. Rakenduse versiooni väljalase.

Küsimus	**KK**
17.Mergetud kood võiks kohe minna testkeskkonda (*Stage*) ja hiljem toodangu keskkonda(*Production*) juhul, kui kõikide automaattestide tulemused on rohelised	4,50
18.Automaatsed infoteated, mis sisaldavad rakenduse väljalaske koosseisu, annavad parema ülevaate väljalaskest huvitatud osapooltele ning on seetõttu väärt kuluvat aega.	4,00
19.Automaatse süsteemi poolt juhitud väljalaske graafik aitab tagada rakenduse paremat kvaliteeti	3,17
*21.*Kanaarjuurutuse võimalus (esialgu teha rakenduse uue versiooni kättesaadavaks ainult teatud % kasutajatest) *CD pipeline-s* aitaks vähendada potentsiaalsete vigade arvu ning on seetõttu väärt kuluvat aega.	4,33

CD – pidevvalmidus/-juurutus, KK – kaalutud keskmine

Saadud tulemustest võib järeldada, et enamik vastajatest toetavad väljalaskmise automaatiseerimise praktikat, nagu pidevjuurutus, autotestide roheliste tulemuste puhul (kaalutud keskmine – 4,5). Küsitlus näitas, et kanaarjuurutuse võimalus sai päris kõrge hinnangu (kaalutud keskmine – 4,33). Positiivse hinnangu said automaatseid infoteateid väljalaske koosseisu kohta (kaalutud keskmine - 4,0) ning väljalaske automaatne juhtimine sätestatud graafiku alusel (kaalutud keskmine – 3,1).

Ploki prioriteedid on järgmised:

- *Merge*-tud kood võib minna toodangusse juhul kui kõik automaatsed kontrollid on läbitud
- Kanaarjuurutus
- Automaatsed infoteated väljalaske kohta
- Automaatne väljalaske graafik

Allolev tabel [Tabel 4] illustreerib küsitluse vastajate arvamust plokis „Monitooring“:

Tabel 4. Monitooring.

Küsimus	**KK**
22.Paindlikumalt konfigureeritavad monitooringu süsteemid aitaksid jälgida rohkem meetrikaid, mis on adopteeritud just meie rakenduste jaoks	4,42
23.Mitme erineva *CI,CD,CM* vahendi kasutamine tagab parima lahenduse arendusprotsessi jaoks	3,17
24.Monitooringu pidev jälgimine arendustiimi poolt tööajal, tõstab rakenduse kvaliteeti ning on seetõttu väärt kuluvat aega.	4,25
25.Monitooringu süsteemid peavad ise informeerima probleemidest (*push alerts*)	4,83
26.Monitooringu süsteem peab kohe andma maksimaalselt informatsiooni probleemi olemusest	3,67
27.Monitooringu süsteemide killustatus/paljusus aeglustab probleemi uurimist	3,67
28.Töövälisel ajal peab monitooring rakenduse vigade kõrge taseme (vigade tase >= 3%) korral informeerima administraatoreid	3,67
29.Vigade kõrge taseme (vigade tase >= 2%) esinemisel, peab *CI/CD* server oskama teha versiooni automaatse tagasikeeramise (ingl k *rollback*)	4,25
30.Monitooringu süsteemid võiksid osata teha enesediagnostikat ja võimalusel ka eneseparandamist	3,58

CI – pidevintegratsioon, *CD* – pidevvalmidus/-juurutus, *CM* – pidevmonitooring, KK – kaalutud keskmine

Saadud tulemuste alusel on näha, et kõige rohkem toetati monitooringu süsteemide automaatset infoteadet (kaalutud keskmine – 4,83). Järgmisena kõrge hinnangu said paindlik monitooringu süsteem, mida on võimalik konfigureerida oma rakenduse vajaduste järgi (kaalutud keskmine – 4,42). Palju vastajaid vastasid, et kõrge vigade taseme korral võib välja kutsuda automaatset tagasipööret eelmisele versioonile (kaalutud keskmine 4,25). Monitooringu pidev jälgimine oli ka kõrgelt hinnatud (kaalutud

keskmine – 4,25). Järgi tulevad sellised ettepanekud nagu maksimaalne informatsioon, mida edastab monitooring ja rakenduse alarmid töövälisel ajal (mõlemal kaalutud keskmine – 3,67). Küsitlus näitas, et paljud vastajad arvavad, et monitooring võib teostada enesediagnostikat ja eneseparandamist (kaalutud keskmine 3,58). Samuti enamus vastajatest vastasid, et monitooringu süsteemide paljusus võib aeglustada probleemi uurimist (kaalutud keskmine 3,67)

Ploki prioriteedid on järgmised:

- Monitooringu automaatsed infoteated (ingl k *push-arerts*)
- Paindlik, konfigureeritav monitooring
- Automaatsed tagasipöörded eelmisele versioonile
- Monitooringu pidev jälgimine
- Maksimaalselt informatiivsed monitooringu infoteated
- Väga kriitilise vigade taseme korral alarmid töövälisel ajal
- Võimalusel minimiseerida monitooringu süsteemide arvu
- Rakenduste enesediagnostika ja eneseparandamine monitooringu abil

Küsitluse viimases plokis oli pakutud kaks küsimust, millele oli võimalik vastata vabas vormis:

- Mida, teie arvates, pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsess võiks veel võtta kasutusele?
- Millest, teie arvates, pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsess võiks loobuda?

Autor tegi kokkuvõtte saadud vastustest ning esimesele küsimusele vastused olid järgmised:

- ülesehitus ja juurutus peavad olema konfigureeritavad koodist, mitte pidevintegratsiooni ja -juurutuse süsteemi kasutajaliidesest;

- mahukate arenduste jaoks oleks kanaarjuurutust kindlasti abiks;
- pidevintegratsiooni ja -juurutuse server võiks osata teha automaatset tagasipööret eelmisele versioonile;
- manuaalse monitooringu osa peab olema minimaalne. Selle asemel peavad olema väga usaldusväärsed automaatsed monitooringu infoteated (ingl k *push-alerts).*

Teisele küsimuse vastuste kokkuvõte näeb järgmiselt välja:

- minimiseerida inimfaktorit monitooringus;
- loobuda monitooringu süsteemide paljususest;
- vähendada staatilise analüüsi kõrged nõudmised;
- vähendada või üldse loobuda *mock-itud backend* testidest.

Küsitlusest saadud informatsiooni põhjal sai autor teada tendentsidest ja planeeris nimekirja töödest, mida ta analüüsib ja realiseerib antud lõputöö raames. Lisaks autor lõi nimekirja jätkutöödest, millega ta näeb vajadust jätkata.

2.4 Tänased pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessid *Reservation* rakenduse baasil

Reservation on rakendus, mis on loodud mikroteenuste arhitektuuri põhjal. Rakendus tegeleb tagasihelistamise ning esindusse aja broneerimisega, mida võivad teostada nii Telia kliendid, kui ka teenindajad klientide nimelt. Lisaks sellele rakenduse administreerimise osa laseb teenindajal tühistada tehtud broneeringuid, märkida, et klient on kohal, muuta informatsiooni esinduste kohta ja lahtioleku aja kohta. Rakendusel eksisteerib *frontend* osa, mis on kättesaadav nii välis, kui ka siseklientidele nimega *Customer Interaction Frontend*. *Backend* osa koos adminliidesega kannab nime *Reservation*. Rakendus on *Stateless*, mis tähendab, et kõik vajalik informatsioon sessiooni kohta antakse kaasa iga pöördumisega.

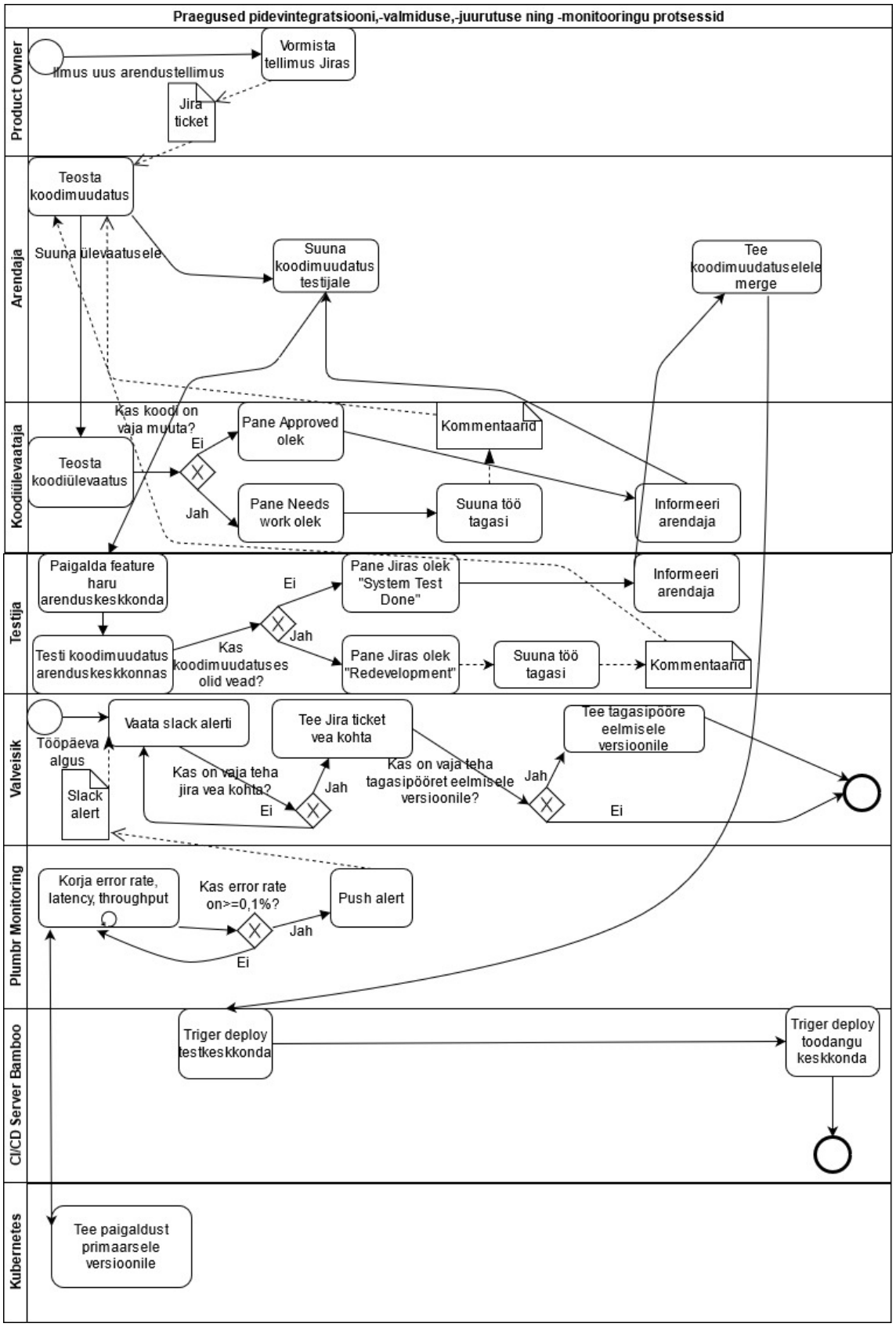
Allolev tabel [Tabel 5] näitab autori poolt teostatud *Reservation* rakenduse praeguse seisu analüüsi eelmises peatükis saadud kriteeriumite alusel:

Tabel 5. Praeguse pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesside seis.

Kriteerium	**Praegune seis**	**Mõju**
Sõltuvate mikroteenuste lokaalselt arendamine ühes komplektis	Praegu ei ole	*Reservation* miktoteenust arendatakse integratsioonis sõltuvate rakenduste ebastabiilsete harudega
Koodi staatiline analüüs koos paranduse ettepanekutega	Praegu ei ole	Koodi süntaks ei ole ühtne
Koodi testidega kaetuse analüüs	On olemas. *Java* poolt teeb staatilist analüüsi *Jacoco*	Koodi kaetus testidega paraneb
Mock-itud ühiktestid *frontend* ja *backend* jaoks	On olemas	Teostavad regressiooni funktsioonide tasemel
Integratsiooni testid teenuste tasemel	On olemas	Teostavad regressiooni moodulite vahel
Vastuvõtmise integratsiooni kasutajaliidese autotestid	Ei ole olemas	Keegi arendustiimist peab teostada manuaaltestimist
Regressiooni kasutajaliidese autotestid	Ei ole olemas	Kasutajaliidese regressiooni testimist on vaja teostada manuaalselt
Võimalus testida kasutajaliidese autotestidega sõltuvaid mikroteenuseid ühes komplektis	Ei ole olemas	Testid piirduvad ainult oma mikroteenuse piiridesse, kuigi funktsionaalne jada võib olla laiem
Koormustestid (*Performance tests*)	Ei ole olemas	Raske on määrata teenuse vastuste viivitust (ingl k *latency*) ja teenuste optimaalset kiirust ilma koormustestideta
Võrdluse (ingl k *Tap-comparison*) testid	Ei ole olemas	Ei anna võimalust testida toodangu keskkonna reaalse info analoogi alusel
Konfiguratsiooni testid	On olemas	Selleks on Test keskkonna (Stage) paigalduse katse
Rakenduse turvalisuse autotestid ja skannerid (*Penetration tests*)	On olemas globaalsel ja lokaalsel tasemel.	Annavad informatsiooni rakenduse turvanõrkuste kohta (ligipääsude testid)
Kanaarväljalaske strateegia	Ei ole olemas	Ei ole võimalus uut versiooni turvaliselt välja lasta
Lippudega väljalaske strateegia	Ei ole olemas	Ei ole võimalust turvaliselt teha lahti ega panna kinni uusi arendusi toodangus

Ootuste konfigureerimine (*SLA, SLO, SLI*)	Ei ole olemas	Ilma selleta on raske püstitada monitooringu põhinäitajaid nagu vigade tase, teenuse vastuste viivitus, tootlikus (ingl k *error rate*, *latency*, *throughput*)
Tehniliste vigade logimine *(*ingl k *Exception tracking)*	On olemas	Vigased olukorrad jäävad märkamata
Automaatsed testid toodangu keskkonnas (*Production Smoke tests*)	Ei ole olemas	Rakendust ei saa testida toodangu konfiguratsiooniga
Profileerimine (ingl k *Profiling)*	On olemas	Raske on uurida mitte ilmselgete probleemide põhjuseid
Päringute jälitus logides *(*ingl k *Tracing)*	On olemas	Saab jälgida päringuid sõltuvates süsteemides
Lõppkasutajate käitumise jälgimine	On olemas	Saab korjata statistikat reaalsete kasutajate käitumismustrite alusel *Google analytics* abil
Rakenduse tehniline tugi	On olemas	Arendustiim saab pakkuda tuge oma lõppklientidele ning teada saab teada kasutusjuhtumitest
Monitooringu meetrikate määramine/konfigureerimine	Ei ole olemas	Meetrikad vajavad rohkem selgust
Alerdid tööajal	On olemas	Arendustiim saab kohe teada rakenduse uutest vigadest ja vigade tasemest
Alarmid töövälisel ajal	Ei ole olemas	Töövälisel ajal probleemidega keegi ei tegele
Automaatne tagasipööre eelmisele versioonile	Ei ole olemas	Rakenduse probleemne versioon võib olla liiga kaua lõppkasutajatele kättesaadav
Süsteemi enesediagnostika ja eneseparandamine	Ei ole olemas	Monitoring nõuab liiga palju inimeste aega

Allolev diagramm [Joonis 4] illustreerib praeguseid pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesse:



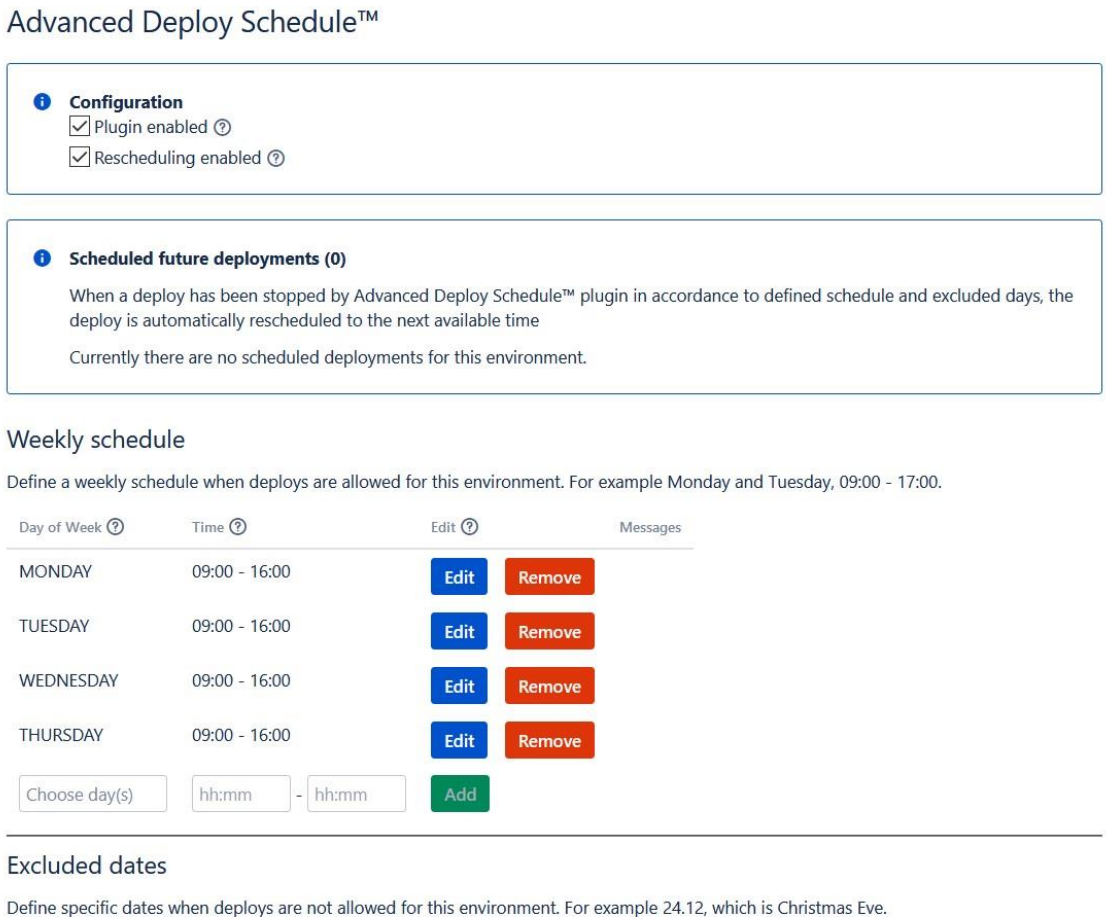
Joonis 4. Tänased pidevintegratsiooni, -valmiduse, -juurutuse ning monitooringu protsessid ühe arendustöö näitel.

Antud skeemil [Joonis 4] on näha, et pärast seda, kui uue arenduse *Jira* tellimuse raames arendaja teeb koodimuudatust ja antud muudatus jõuab koodihoidlasse, käivituvad pidevintegratsiooni ja -juurutuse serveril ühik-ja integratsioonitestid. Pärast autotestide edukat läbimist arendaja loob *pull-request'i* ja lisab arendustööle ülevaatajaid. Juhul kui koodimuudatus läbib koodiülevaatuse faasi, seda suunatakse testijale manuaalseks testimiseks. Selles faasis teostatakse uue arenduse muudatuste ja regressiooni teste. Regressiooni testid tavaliselt võtavad aega umbes kaks tundi.

Kui arendustöö läbib manuaalse testimise etappi, testija paneb *Jira* tellimusse staatuse „*System Test Done*". Pärast seda arendaja teeb antud koodi muudatusele *merge master* haruga. Antud etapil *master* harus käivituvad ühik-ja integratsiooni testid. Automaatseid regressiooni kasutajaliidese teste selles etapis ei tehta. Juhul kui ühik -ja integratsiooni testide jooksutamine läheb läbi ilma vigadeta, pidevintegratsiooni ja -juurutuse server loob *Docker* konteineri. *Docker* konteineri ülesehitus kutsub välja juurutust testkeskkonda (*Stage*) , mis omakorda eduka juurutuse korral käivitab juurutust toodangu keskkonda (*Production*). Selleks, et uued arendused ei läheks toodangu keskkonda töövälisel ajal, pidevjuurutuse protsessi on integreeritud programm, mis juhib juurutuse graafikut – *Advanced Deploy Schedule*.

Allolev joonis [Joonis 5] illustreerib seda kuidas näeb välja pidevjuurutuse süsteemi lisa – *Advanced Deploy Schedule*, mis oli arendatud Telia Eesti AS arendajate poolt *Hackathon*[1] ürituse raames:

[1] Hackathon – üritus, mille raames arendajad, disainerid, arendusjuhid ja teised lahendavad koos mingi probleemi piiratud ajavahemikul



Joonis 5. Bamboo Advanced Deploy Schedule.

Antud programm aitab vähendada riske, mis on seotud automaatse monitooringu puudumisega.

Pärast seda kui rakenduse uus versioon jõuab toodangu keskkonda on arendajal vaja jälgida rakenduse seisu monitooringu süsteemis, mis agregeerib logisid - *Graylog*. Lisaks valveisik, kelleks on arendustiimi roteeruv liige, koguaeg jälgib *Slack Messengeri* monitooringu kanalit.

Monitooringu süsteemides antud hetkel aktiivselt kasutusel on kaks süsteemi: *Graylog*, mis on logide agregaator, ja *Plumbr*, mis on funktsionaalse monitooringu valmistoode, kus saab konfigureerida alerte. *Graylog* monitooringus saab vaadata seoseid erinevate päringute vahel *X-Request-id* järgi juhul kui nad olid tehtud sama sessiooni raames. *Plumbr* monitooring pakub vea korral vaadata logide jälitamist (ingl k *Tracing*), mis tähendab, et saab teada mis toimub enne ja pärast vigast olukorda.

Valveisik jälgib *Slack Messengeri* kanalit ning monitooringu süsteeme – *Graylog* ja *Plumbr*, tööpäeviti kella üheksast kuni viieni. Töövälisel ajal rakenduse seisu ei jälgita.

2.4.1 *Plumbr* konfiguratsioon praegustes protsessides *Reservation* rakenduse baasil

Plumbr on monitooringu süsteem, mida kasutatakse *Reservation* rakenduse funktsionaalseks monitooringuks. Hetkel *Plumbr* süsteemi võib pidada monitooringu põhisüsteemiks *Reservation* rakenduse jaoks.

Plumbr monitooringu süsteemi *Reservation* rakendusele oli määratud *SLO* ja selle alusel olid ümber konfigureeritud meetrikate väärtuste piirid.

Meetrikad, mis olid defineeritud:

- vigade tase (ingl k *error rate*);
- latentsus (ingl k latency);
- tootlikus (ingl k *throughput*).

Oli otsustatud, et lähtudes *Reservation* rakenduse kriitilisusest ning tarbijate arvust, vastavad meetrikate piirväärtused võivad olla:

- *Reservation* rakenduse *SLO* – mitte madalam kui 99,9% ;
- vigade tase – mitte kõrgem kui 0,1%;
- vigade eelarve (ingl k *error budget*) 30 päeva jooksul ei saa ületada 0,1%;
- latentsuse mediaan – mitte kõrgem kui 1000ms;
- 99. protsentiil – mitte kõrgem 1000ms
- tootlikus – mitte kõrgem kui 1000 teenuse väljakutsumist 10 minuti jooksul

Juhul kui reaalsed *SLI*-d ületasid piire, monitooringu süsteem *Plumbr* saadab *push* infoteadet (ingl k *alert*) *Slack Messenger* arendustiimi monitooringu kanalisse. Antud kanal on jälgitav valveisiku poolt, kelleks on arendustiimi liige. Infoteate alusel otsustatakse kas tegu on kriitilise probleemiga või mitte.

Põhjus, miks vigade tase oli pandud tasemele 0,1%, seisneb selles , et erinevalt *frontend* rakendusest, *backend* rakendus peab olema maksimaalselt kättesaadav ja vastama *SLO*-le 99,9%.

2.5 Uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessid *Reservation* rakenduse baasil

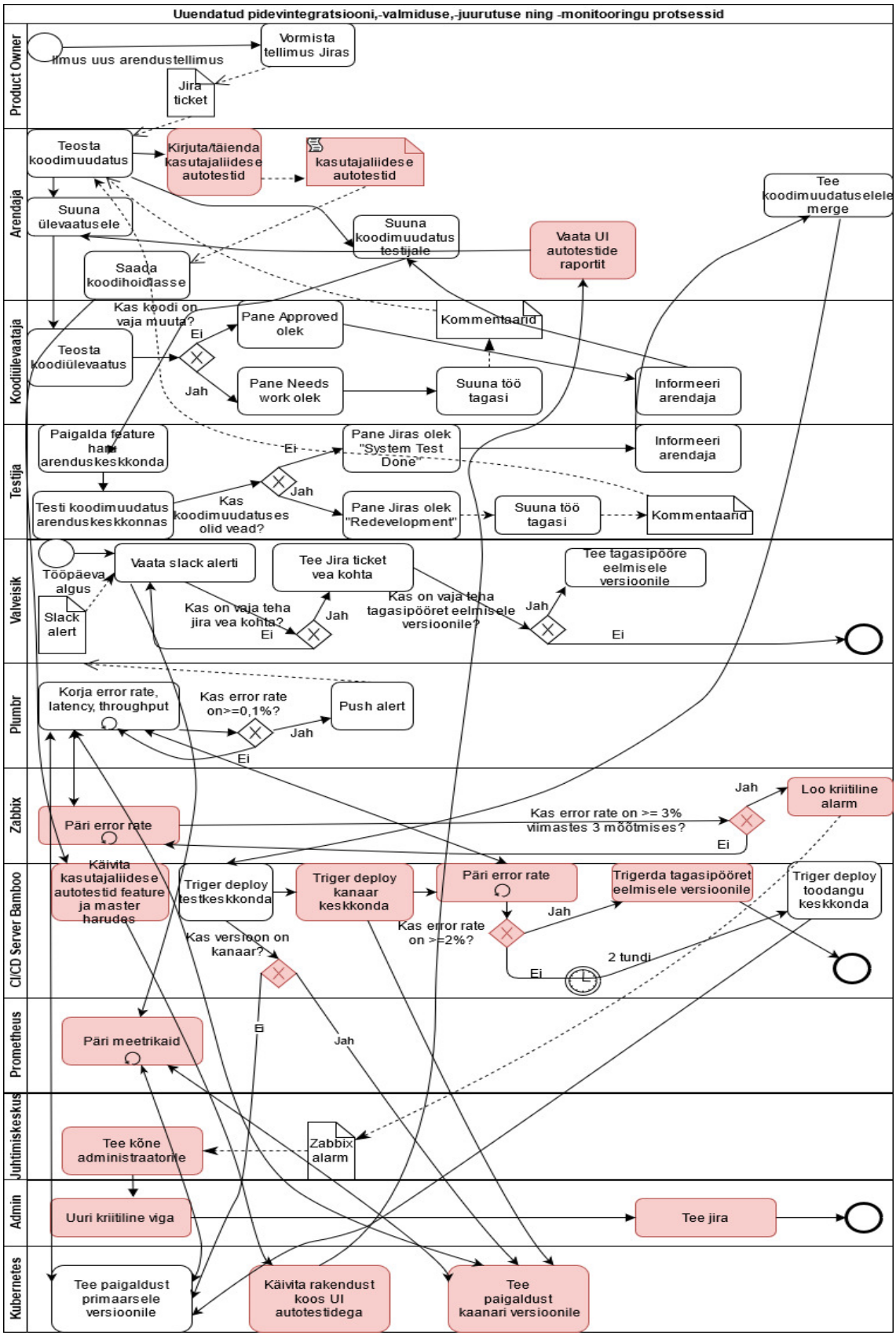
Lähtudes pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside maailma IT praktikatest, küsitlusest saadud vastustest ning *Reservation* rakenduse teostatud revisjonist planeerisin nende protsesside parandamiseks teha ettepanekud, mis on väljatoodud allolevas tabelis [Tabel 6]:

Tabel 6. Tulevaste pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside mõju ja teostuse aeg.

Kriteerium	**Mõju**	**Skoop**
Võimalus lokaalselt arendada sõltuvaid mikroteenuseid ühes komplektis	Arendada ja testida saab kiiremini integratsioonis õigete harudega sõltuvates rakendustes	Ootab arutelu kuna veel ei ole saanud tuge. Potentsiaalne jätkutöö
Vastuvõtmise kasutajaliidese integratsiooni autotestid	Tulevikus võib loobuda manuaalsest testimisest	Ootab arutelu kuna veel ei ole saanud tuge. Potentsiaalne jätkutöö
Regressiooni kasutajaliidese autotestid	Suuri koodimuudatusi saab teostada kindlamalt	Antud lõputöö skoop
Võimalus automaattestidega testida sõltuvaid mikroteenuseid ühes komplektis	Testida saab funktsionaalsust terves jadas	Ootab arutelu, potentsiaalne jätkutöö
Koormustestid (*Load tests*)	Saab täpsemini määrata teenuste vastuste viivitust ja teenuste vastuste aega	Jätkutöö
Kanaarjuurutus	Rakenduse uut versiooni saab juurutada järk-järgult ja turvaliselt	Antud lõputöö skoop
Testid toodangu keskkonnas (*Production Smoke tests*)	Toodangu keskkonnas saab kindluse jaoks teostada kontrolltestid	Jätkutöö
Monitooringu meetrikate määramine/konfigureerimine	Meetrikad on loodud süsteemi vajaduste järgi	Antud lõputöö skoop

Alarm-id töövälisel ajal	Kõrge vigade tase ei jää märkamatuks töövälisel ajal	Antud lõputöö skoop
Automaatne *tagasipööre* eelmisele versioonile	Vähem kliente saavad olla mõjutatud probleemse versiooni tõttu	Antud lõputöö skoop

Allolev diagramm [Joonis 6] illustreerib uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesse, roosa värviga on määratud protsesside osad, mida autor realiseerib antud töö raames:



Joonis 6. Uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessid ühe arendustöö näitel.

Antud skeemil [Joonis 6] on näha, et erinevalt vanast protsessist, pärast seda kui koodimuudatus jõuab koodihoidlasse, pidevintegratsiooni ja -juurutuse serveril käivituvad regressiooni kasutajaliidese autotestid. See lähenemine annab võimaluse loobuda manuaalsest regressiooni testimisest ning annab arendajale kiiremat tagasisidet juba enne koodiülevaatust. Juhul kui koodiülevaatest ja manuaalsest testimisest ei tule vajadust koodi parandada, arendaja teeb koodimuudatusele *merge master* haruga. Pidevintegratsiooni ja -juurutuse serveril jälle käivituvad regressiooni kasutajaliidese autotestid.

Juhul kui kõik automaattestid *master* harus läksid läbi ilma vigadeta, pidevjuurutuse server automaatselt alustab selle juurutuse paki paigaldust testkeskkonda (*Stage*). Juhul kui testkeskkonda paigaldus on edukas, sama juurutuse pakk läheb kanaarkeskkonda (*Canary*). See tähendab, et toodangu keskkonda luuakse rakenduse eksemplarid koos uue versiooniga, samal ajal rakenduse eksemplarid koos vana versiooniga jäävad alles. Koormusejaotur suunab umbes kakskümmend protsenti võrguliiklusest uuele versioonile ja kaheksakümmend protsenti vanale. Pärast kanaarjuurutust pidevintegratsiooni ja -juurutuse serveril käivitud skript, mis korjab informatsiooni *Plumbr* monitooringu süsteemist vigade taseme kohta projektis „reservation-canary“. Antud etapp kestab kaks tundi ja juhul kui selle etapi jooksul vigade tase tõuseb kuni kahe protsentini, skript kutsub välja pidevintegratsiooni ja -juurutuse süsteemi tegema *Reservation* rakenduse versiooni tagasipööret eelmisele versioonile. Juhul kui vigade tase kahe tunni jooksul ei tõusnud kõrgemale kui kaks protsenti, pidevintegratsiooni ja -juurutuse server kutsub välja skripti, mis paigaldab kanaarversiooni kõikidele rakenduse eksemplaridele ja antud versioon muutub primaarseks.

Pärast primaarse versiooni juurutust pidevintegratsiooni ja -juurutuse serveril käivitub skript, mis korjab informatsiooni *Plumbr* monitooringu süsteemist vigade taseme kohta projektis „reservation“. Antud etapp kestab kaks tundi ja juhul kui selle etapi jooksul vigade tase tõuseb kuni kahe protsentini, skript kutsub välja pidevintegratsiooni ja -juurutuse süsteemi *Reservation* rakenduse versiooni tagasipööret eelmisele versioonile.

Samamoodi nagu ka vanas protsessis, jääb uuendatud protsessi *Slack Messengeri* monitooringu kanali jälgimine, kuhu saadab *alerte* monitooringu süsteem *Plumbr*, tööpäeviti kella üheksast kuni viieni. Töövälisel ajal rakenduse seisu valveisik ei jälgi.

Lisaks skeemil [Joonis 6] on näha, et *Reservation* rakendusele autor loob integratsiooni *Prometheus* monitooringu süsteemiga, mis annab võimalust paindlikumalt konfigureerida meetrikaid ja luua nende alusel hiljem informatiivseid graafikuid analoogselt *Plumbr* monitooringu süsteemile. *Prometheus* kasutuselevõtt annaks võimalust tulevikus loobuda *Plumbr* monitooringust vajaduse korral.

Skeemil [Joonis 6] on ka näha, et tekib integratsioon *Zabbix* monitooringu süsteemiga, mis hakkab jälgima vigade taset *Plumbr* monitooringus. Juhul kui vigade tase on kõrgem kui kolm protsenti luuakse *alarm Juhtimiskeskusesse*. *Juhtimiskeskus* omakorda helistab *Reservation* rakenduse administraatorile, kes hakkab uurima probleemi ja otsustab mida saab ette võtta.

Tuleviku perspektiivis see toob KPI (ingl k *Key Performance Indicators*) muudatusi, nimelt:

- Arenduse aeg (ingl k *Development Time*) (Arenduse võtmise hetkest kuni väljalaskeni koos tunnikestva monitooringuga) väheneb tänu lisatud kasutajaliidese integratsiooni testidele.

- Aeg uue arenduse väljalaskmiseni (ingl k *Time to Market*) langeb tänu lisatud kasutajaliidese integratsiooni testidele ja kiire tagasiside tsüklile kanaarjuurutuse ja monitooringu kaudu

- Toodangu vigade arv (ingl k *Critical and Major Live Bugs Count*) CRM valdkonna rakendustes langeb tänu kasutajaliidese integratsiooni automaattestidele ja kanaarjuurutamisele

- Kriitiliste vigade keskmine lahendamise aeg (ingl k *Critical Bugs Average resolution Time*) langeb tänu täiendatud monitooringule *Plumbri*, *Prometheus* ja *Zabbix* näol. Lisaks kriitiliste vigade vähendamist toetab ka versiooni automaatne tagasikeeramise võimalus.

2.5.1 Uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside puudujäägid

Kuigi uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsessid laiendavad arendusprotsessi võimalusi, kõik selle lülid ei ole praegu kõige optimaalsemad kuna reaalne asjaolu piirab tehnoloogiate.

Kasutajaliidese autotestide puudujäägina võib välja tuua seda, et nad suurendavad paki ülesehitamise aega ning neid on vaja aeg-ajalt täiendada ja parandada. Aga lisaväärtus mida kasutajaliidese autotestid annavad regressiooni kontrollile on kindlasti suurem. Lisaks kasutajaliidese autotestid on elav dokumentatsioon, mis ajaga muutub aina aktuaalsemaks agiilsete arenduste projektides, kus tavaliselt dokumentatsiooni eraldi ei looda. Lisaks praktika näitab, et kui automatiseerimise tehnoloogia oli valitud õigesti, siis testide haldus ei võta palju aega.

Kanaarjuurutuse strateegia puudujäägina võib välja tuua seda, et juhul kui kanaarversiooniga ilmuvad probleemid, siis tuleb eemaldada terve kanaarjuurutus. Alternatiivina võib välja tuua lippude strateegiat (ingl k *feature flags*). See annab võimalust avada või sulgeda uute arenduste kättesaadavust lipu praeguse väärtuse alusel. Antud tehnika võimaldab kontrollida väljalaset otse koodist, kuna arendajal on vaja lisada *if-else* lauseid selleks, et defineerida koodilõiku, mis peab olema juhitav lipuga. Lisaks lipu väärtust peab kuskil hoidma. Seda võib teha mitmel viisil: süsteemiga, mis hoiab lippude väärtuseid, andmebaasis, konfiguratsiooni failis ja nii edasi. Autori arendustiimis kasutatakse ka praegu aeg-ajalt antud tehnikat, andes üle lippude väärtusi koos rakenduse konfiguratsiooni failiga. Autori arvamusel, lippude tehnikal on teistsugune tööpõhimõte võrreldes kanaarjuurutusega: lippude tehnika kontrollib kas uus funktsionaalsus oli õigesti projekteeritud ja programmeeritud, kuid kanaari tehnika kontrollib kas uus rakenduse versioon ei riku olemasolevat rakendust. Mõlemad tehnikad on head ja teenivad oma eesmärke.

G. Schermann, J. Cito ja P. Leitner [20] tegid uuringu, mis näitas, et arendustiimide arvamusel, kõige efektiivsem on kasutada eksperimentides nii kanaarjuurutust kui ka lippude strateegiat, kus kanaarjuurutus toetaks regressiooni eksperimente ja lippude strateegia – äriekperimente. Kanaarjuurutuse analüüsi puudujääkidena autorid toovad *CPU* ressursside ja läbilaskvuse (ingl k *throughput*) kasvu kuna kanaarjuurutus loob rakenduse lisa eksemplare. Lisaks autorid pööravad tähelepanu sellele, et organisatsiooni

infrastruktuur ja arendustiimid peavad olema valmis selleks, et implementeerida antud lahendusi ja teostada statistilist analüüsi monitooringust saadud andmete alusel. Lisaks neid eksperimente peavad toetama organisatsiooni äri esindajad.

Prometheus meetrikate logimine võib olla mingil määral dubleeritud *Plumbr* süsteemi poolt ja sellisel juhul osaliselt dubleeritavate süsteemide üleval hoidmine ei tundu mõistlik. Aga juhul kui arendustiim valib Prometheuse, siis alustada üles ehitada teist monitooringu süsteemi on mõistlik juba praegu.

2.6 Tänaste ja uute pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesside erinevused

Pärast uuendatud protsessi kasutusele võtmist *Reservation* rakenduses ilmuvad kasutajaliidese integratsiooni testid, mis hakkavad käivituma pidevintegratsiooni ja-juurutuse serveril rakenduse koodi iga ülesehituse protsessiga, igas harus. Antud uuendus toob kaasa toodangu keskkonnas vigade arvu langust (*KPI*: Kriitiliste ja oluliste vigade arv toodangu keskkonnas), arenduse aja vähenemist (*KPI*: Arenduse aeg) ning vähendab aega uue funktsionaalsuse väljalaskmiseni (*KPI*: Aeg uue funktsionaalsuse väljalaskmiseni).

Uuendatud protsessis ilmub võimalus teostada kanaarjuurutust, mille käigus ainult teatud protsent kasutajatest saab ligi rakenduse uuele versioonile. See strateegia minimiseeriks potentsiaalsetest vigadest mõjutatud kasutajate arvu, mis omakorda toob kaasa selliste *KPI*-de vähendamist nagu Kriitiliste ja oluliste vigade arv toodangu keskkonnas, Arenduse aeg.

Lisaks uuendatud protsessis ilmub integratsioon uue monitooringu süsteemiga – *Prometheus*, mis hakkab korjama lisa meetrikaid *Kubernetes Pod*-idest ja hiljem looma nende alusel konfigureeritavaid graafikuid. *Prometheus* on väga tõrkekindel süsteem, mis oskab töödelda kolossaalset meetrikate mahtu. Antud integratsioon annab võimalust ka tulevikus kasutada ainult *Prometheus* monitooringut koos tema integratsioonidega. Antud uuendused toovad kaasa toodangu keskkonnas vigade arvu langust (*KPI*: Kriitiliste ja oluliste vigade arv toodangu keskkonnas) tänu rakenduse täpsemale analüüsile. Lisaks tänu relevantsele monitooringule langeb kriitiliste vigade lahendamise aeg.

Juhul kui vigade tase tõuseb liiga kõrgele tasemele (>=2%), uuendatud protsessis on olemas võimalus teostada automaatset tagasipööret rakenduse eelmisele versioonile. Antud tehnikat võib kasutada nii kanaari kui ka primaarse väljalaskmisega. Antud muudatus vähendab toodangu vigade arvu (*KPI*: Kriitiliste ja oluliste vigade arv toodangu keskkonnas).

Uuendatud protsessis, monitooringu süsteem *Zabbix*, jälgib 24/7 *Reservation* rakenduse vigade taset ja väga kõrge vigade taseme korral (>=3%) loob kriitilise alarmi *Juhtimiskeskusele*, kes omakorda helistab rakenduse administraatoritele. Antud muudatus toob kaasa toodangu keskkonnas vigade arvu langust (*KPI*: Kriitiliste ja oluliste vigade arv toodangu keskkonnas). Lisaks langeb kriitiliste vigade lahendamise keskmine aeg (*KPI*: Toodangu keskkonda vigade keskmine lahendamise aeg).

3 Uus realisatsioon

Käesoleva peatüki eesmärk on kirjeldada uuendatud pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside realisatsiooni. Tuua argumente miks olid valitud just need tehnoloogiad ning kuidas võib realiseerida samu ülesandeid teisiti. Lisaks autor toob välja jätkutööde nimekirja, mida oleks vaja realiseerida tulevikus.

3.1 Automaattestimine

Reservation rakenduses on juba realiseeritud ühiktestid *frontend* ja *backend* plokkide jaoks. Autori arvamusel teema, mis vajab praegu tähelepanu on kasutajaliidese regressiooni automaattestid.

3.1.1 Kasutajaliidese automaattestid

Selleks, et *Reservation* rakendus vastaks efektiivsele protsessile oli vaja realiseerida rakenduse *frontend-is* kasutajaliidese regressiooni automaattestid kõikide kriitiliste kasutusjuhtumite jaoks. Eelnevalt autor analüüsis ja kooskõlastas automaattestide nimekirja *Reservation* rakenduse *Product Owneriga* ning *Product Manageriga.*

Nimekiri sai järgmine:

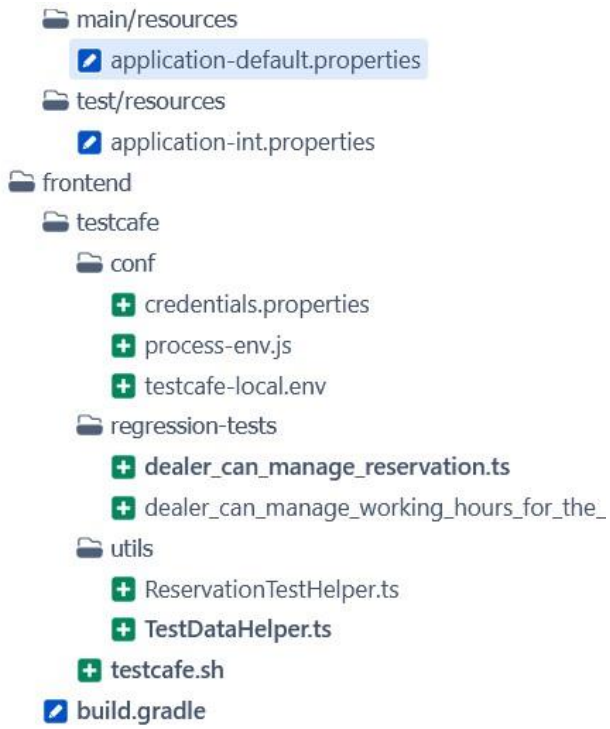
- teenindaja saab vaadata ja tühistada tehtud tagasihelistamise broneeringuid;
- teenindaja saab vaadata ja tühistada tehtud esindusse visiidi aja broneeringuid;
- teenindaja saab hallata esinduse lahtioleku regulaarseid aegu;
- teenindaja saab hallata esinduse lahtiolekuaegu pühade ajal;
- teenindaja saab muuta informatsiooni esinduse kohta.

Kasutajaliidese regressiooni automaattestide tehnoloogiaks autor valis *Testcafe* ning programmeerimiskeeleks *TypeScript.*

Lahenduse algoritm sai olema järgmine:

1. luuakse testide skriptid;
2. luuakse testide Helper klassid;
3. salajased konfiguratsioonid on krüpteeritud kujul;
4. testi käigus Rest-api väljakutse abil genereeritakse testandmeid (reserveeringuid);
5. pidevaintegratsiooni serveril *master* ja *feature* harudes, *Node+Testcafe Docker image* sees käivituvad:
   i) Reservation rakendusel käivitub *backend* ja *frontend* osad;
   ii) Chrome headless ja paralleelses režiimis käivituvad *Testcafe* kasutajaliidese testid;
6. raportit genereeritakse *XML* ja *HTML* formaadis, infoteatega emailile.

Järgmine joonis [Joonis 7] näitab kataloogide ja failide struktuuri:



Joonis 7. Testcafe kasutajaliidese testide struktuur.

Automaattestides kasutatakse Page Object Pattern [21], kus lehe tehnilised detailid nagu *CSS selectorid*[1] ja *DOM*[2] elementide kirjeldused asuvad *Helper* klassides ning test ise näeb välja nagu nimekiri kasutaja tegevustest.

Koodinäited asuvad antud lõputöö plokis Lisad - Lisa 1, nimega „Testcafe kasutajaliidese testid“

3.1.2 Alternatiivsed lahendused testimisega seoses

Sama ülesande lahendamiseks sobib ka teine automaatestimise raamistik – *Cypress*. Sarnaselt *Testcafe* raamistikuga *Cypress* ka käivitub osaliselt brauseris ja osaliselt Node.JS-s [22]. Mõlemad raamistikud on väga töökindlad ja oskavad teostada testi kontrolle (ingl k *assertions*) mitu korda, mis vähendad testide ebastabiilsust. Kuid *Testcafe* raamistikul on brauserite tugi. Teste võib käivitada kõikides brauserites seal hulgas ka integratsioonis *BrowserStack* ja *SaurceLabs* platvormidega. Aga lõpliku valiku *Testcafe* kasuks aitas teha ikkagi see, et hetkel see on kasutusel ka teistes rakendustes autori arendustiimis. *Testcafe* raamistik sobis antud automaattestide komplekti realisatsiooni jaoks, kuid autor näeb jätkutööde vajadust. Jätkutööna siinkohal võib välja tuua realisatsiooni sõltuvate rakenduste kooskäivitamist, muutes selle jaoks pidevintegratsiooni süsteemi konfiguratsiooni, niimoodi, et sõltuvate süsteemide koodi hoidlad ja *Docker imaged* oleksid ühes *build* plaanis. See konstruktsioon tagaks kõige õigemat sõltuvate rakenduste versioonide komplekti.

Lisaks autor näeb vajadust realiseerida autotestid toodangu keskkonnas (*Smoke tests*), mis hõlmaksid ainult ärikriitilist loogikat. Antud testid kasutaksid toodangu keskkonna olemasolevaid era ja äritestkliente.

Koormustestid on testide liik, millest tihti mõeldakse, aga harva realiseeritakse, kuna nende realiseerimise kulu on üsna kõrge. Kuigi nende testide kasutamine toodangu keskkonnas tooks suure kasu süsteemi võimaluste tundmise aspektis. Seega antud testide liik sai ka jätkutööde nimekirja.

[1] Elemendid, mis otsivad ja stiliseerivad HTML elemente

[2] Elemendi objekt, mis esindab HTML objekti

3.1.3 Järeldused testimisega seoses

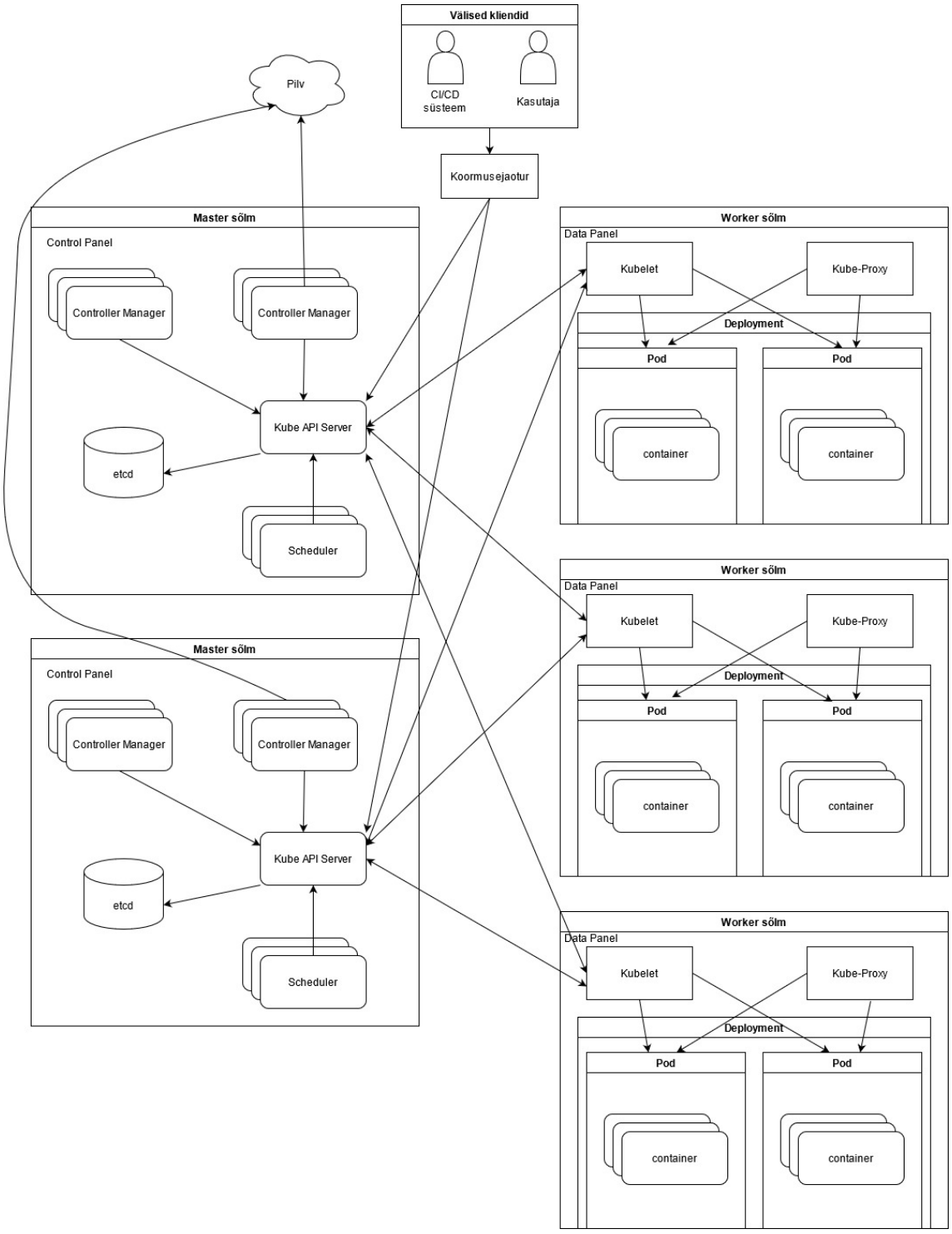
Regressiooni autotestide kontekstis antud tehnoloogia ja arhitektuuriline lahendus on sobilikud, sest *TypeScriptil* on üsna mugav süntaks, *Testcafe* raamistikul on olemas kõik enamlevinud funktsionaalsused, mis võivad olla vajalikud rakenduse kasutajaliidese testimises nagu autentifikatsioon modaalakende kaudu. Testandmete genereerimine ka toimub just sellel minimaalsel tasemel, mis on vajalik testide teostamiseks.

Autori arvamusel, realiseeritud kasutajaliidese integratsiooni testid lahendavad regressiooni testimise probleemi ja arendustiim ei pea enam eraldi eraldama aega manuaalse regressiooni testimise jaoks *Reservation* koodi suurte muudatuste puhul. Automaatsete regressiooni testide olemasolu omakorda vähendab arenduse aega (*KPI*: Arenduse aeg), vähendab vigade arvu toodangu keskkonnas (*KPI*: Kriitiliste ja oluliste vigade arv toodangu keskkonnas), vähendab aega väljalaskmiseni (*KPI*: Aeg väljalaskmiseni) ning tõstab arendustiimi üldist kindlust rakenduse kvaliteedis.

3.2 Rakenduse uue versiooni kanaarväljalase

Kanaarväljalaske (ingl k *Canary release*) strateegiat võib teostada mitmel viisil ja siin sõltub palju infrastruktuuri tehnilistest võimalustest ning rakenduse vajadustest, nimelt, kas rakenduse jaoks on oluline, et oleks säilitatud konkreetne seis nagu sessioon või rakendus on täiesti vaba sellest nõudmisest.

Reservation rakendus hoiab informatsiooni sessiooni kohta koos iga päringuga ja seega ei vaja eraldi loogikat sessioonide hoidmise jaoks. Rakendus töötab *Docker* konteineris, mis omakorda töötab konteinerite haldamise platvormil – *Kubernetes*. Allolev joonis [Joonis 8] näitab *Kubernetes* arhitektuuri, mis on analoogne sellele, kus töötab *Reservation* rakendus:



Joonis 8. Kubernetes arhitektuur.

Kubernetes infrastruktuuris eksisteerib mitu *master* sõlme ja mitu *worker* sõlme. Võrguliiklus jõuab *master* sõlmedeni koormusejaoturi kaudu (ingl k *Load Balancer*), kust edasi *master* sõlmed suhtlevad *Kube-api* kaudu *worker* sõlmedega *Kubelet* objekti abil. *Kubelet* omakorda juba juhib *Deployment* protsesse *Pod*-idel, kus jooksevad *Docker*

konteinerid rakenduste versioonidega. Igast *Pod*-ist on olemas üks kuni mitu replikatsiooni. Sama loogika järgi töötab ka *Reservation* rakenduse juurutamise loogika.

Antud lõputöö ülesanne on luua võimalus, kus *Kubernetes* platvormis jätkab töötama X arv replikatsioone rakenduse vana versiooniga ja ilmub Y arv replikatsioone rakenduse uue versiooniga – *kanaari*.

Selles töös autor realiseerib kanaarväljalaset, täiendades *Kubernetes Deployment* ja *Service* objekte ja lisades *Nginx* serverit võrguliikluse juhtimiseks.

3.2.1 Kubernetes Deployment ja Service objektide täiendus ning Nginx loomine

Reservation rakenduse praegune *Kubernetes* kataloogi ülesehitus on järgmine:

- k8s_configmap.yaml
- k8s_deployment.yaml
- k8s_secret.yaml
- k8s_service.yaml

Kus ***ConfigMap*** on *Kubernetes API* objekt, mille kaudu võib konfigureerida *Pod* konteinerit, andes üle keskkonna muutujaid nagu on näidatud joonisel [Joonis 9]:

```
data:
  spring.profiles.active: {SPRING_PROFILE}
  application.ci: {APPLICATION_CI}
  plumbr.appname: {PLUMBR_APPNAME}
  plumbr.appname.canary: {PLUMBR_APPNAME_CANARY}
```

Joonis 9. Kubernetes ConfigMap.

Secret on *Kubernetes API* objekt, mille kaudu võib *Kubernetes Pod*-ile üle anda salajaseid informatsiooni nagu, näiteks, *Reservation* rakenduse seadistusi.

Deployment on *Kubernetes API* objekt, mille kaudu *Kubernetes* kontrollib *Reservation* rakenduse seisu *Pod*-ides.

Service on *Kubernetes API* object, mille kaudu *Kubernetes* võib representeerida *Reservation* rakenduse *Pod*-e ühe abstraktsiooni all, ühendades neid sama *selectori* alla.

Kanaarväljalaske strateegia lahenduse algoritm sai olema järgmine:

1. lisada *Deployment* objekt koos *canary* labeliga ja replikatsioonide arvuga kanaari jaoks;
2. lisada *Service* objekt kanaari jaoks (vajalik võrguliikluse suunamiseks);
3. lisada bash skript kubectl käskudega kanaari *Deployment* ja *Service* objektide jaoks.

Detailides seda võib kirjeldada järgmiselt:

Kanaari lahenduse realisatsioonis on vaja luua lisa *Deployment* kontroller, mis lisaksid *Reservation Kubernetes Pod*-ele *selectori* labeliga „canary“ nagu seda pakub M. Lukša [23] ning konfiguratsioon näeb välja järgmiselt [Joonis 10].

```
selector:
  matchLabels:
    app: reservation-spring
    track: canary
```

Joonis 10. Kanaarjuurutuse label.

Pod-ide replikatsiooni spetsifikatsioon saab endale 1 eksemplari nagu on näidatud joonisel [Joonis 11]:

```
spec:
  replicas: 1
```

Joonis 11. Kanaarjuurutuse replikatsioonid.

Lisaks *Kubernetes Deployment* kontrollerist saab ka viidet *Plumbr* kanaari projektile. *Reservation Canary Service* objekt on identne primaarse *Reservation Service* objektiga, erinedes viimasest ainult nimega.

Selleks, et tagada võrguliikluse jaotus, mis on vajalik kanaarjuurutamiseks, oli otsustatud luua veel üks konteiner – *Nginx* [24]. *Nginx* rakendusel on päris palju kasutamise võimalusi. Antud töös autor kasutab seda koormusejaotur (ingl k *Load Balancer)* lahenduseks.

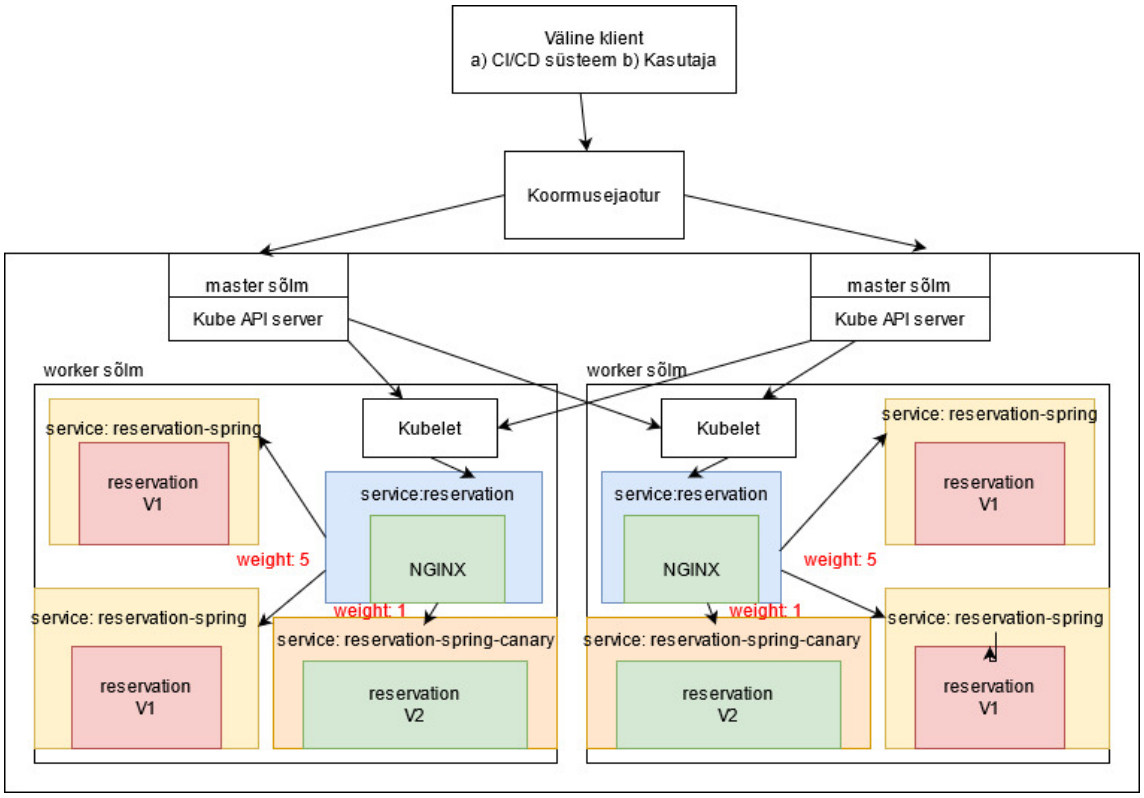
Nginx konteineri loomine sai endale *Reservation* rakendusega sarnase komplekti, mis koosneb *Deployment*, *ConfigMap* ning *Service Kubernetes* objektidest. Lisaks *Nginx* vajab ka konfigureerimise faili, kuhu koormusejaoturi ülesande täitmiseks läksid järgmised read mida kajastab joonis [Joonis 12]:

```
log_format main_uuid '$remote_addr - $remote_user [$time_local] $http_x_request_id '
                     '"$request" $status $bytes_sent '
                     '"$http_referer" "$http_user_agent"';

upstream reservation {

  server reservation-spring max_fails=0 weight=5;
  server reservation-spring-canary max_fails=0 fail_timeout=600
  weight=1;
  keepalive 32;
}
```

Joonis 12. Nginx konfiguratsioon.

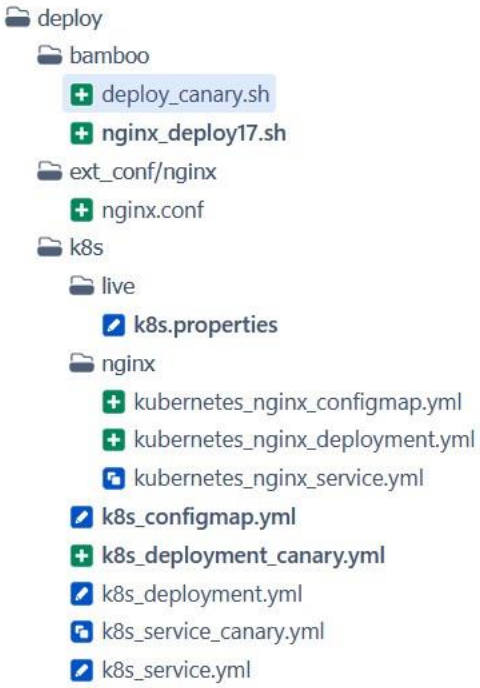
Nginx konfiguratsioon annab kaalu (ingl k *weight*) nii kanaari serveritele kui ka serveritele millel on rakenduse primaarne versioon: 5 ja 1 praegusel juhul.

Allolev joonis [Joonis 13] näitab kuidas näeb välja *Reservation* rakenduse töö kanaarjuurutuse ja -väljalaske raames:



Joonis 13. Rakenduse uue versiooni kanaarjuurutus ja -väljalase Kuberneteses.

Kataloogide struktuur sai järgmiseks [Joonis 14]:



Joonis 14. Kanaarjuurutuse failide muudatused.

Selleks, et teostada kanaarjuurutuse versiooneerimise strateegiat oli loodud *deploymenti* uus *bash* skript, mis kutsub välja *Kubernetes* kontrollerit „reservation-spring-canary“. Antud skripti on vaja välja kutsuda pidevintegratsiooni ja -juurutuse süsteemist. Allolev joonis [Joonis 15] näitab kuidas *Bamboo* serveril näeb välja kaanarjuurutuse etapp:

Joonis 15. Bamboo rakenduse versiooni juurutuse etapid.

Joonisel [Joonis 15] on näidatud 4 sammu rakenduse versiooni väljalaske protsessist:

dev – arenduskeskkond (*Development*);

test – testkeskkond (*Stage*)

canary – kanaarjuurutuse keskkond (*Canary*) (vaikimisi keskkond on sisse lülitatud);

live – toodangu keskkond (*Production*).

Samm „canary“ kutsub välja deploy_canary.sh skripti, mis omakorda kutsub välja *Kubernetes Deployment* just kanaarjuurutuse jaoks.

3.2.2 Alternatiivsed lahendused

Maailmas eksisteerivad väga efektiivsed ja läbimõeldud valmislahendused kanaarläbilaske teostamiseks nagu *Gitlab CI* [25], *Argo Rollouts* [26], *Istio* [28] koos *Flagger* [27] süsteemiga. Ise *Kubernetes* pakub kasutada võrguliikluse suunamiseks *Gloo* [29]. Praegu neid rakendada ei olnud võimalik, kuna see nõuaks terve *Kubernetes* klastri konfiguratsiooni muutmist, millest sõltuvad ka paljud teised rakenduste paigaldused ning

töö. Tulevikus kanaarläbilaske populaarsuse kasvuga, võib-olla saab seda realiseerida juba globaalselt *Kubernetes* klastri tasemel.

Juhul kui rakendus on *Stateful* ehk tema jaoks on oluline sessiooni püsivus ja ta hoiab seda masinal, kus praegu rakendus töötab, siis kanaarväljalaskmise jaoks võib kasutada Kubernetes teist kontrollerit - *StatefulSet*. Sellel juhul *Nginx* koormusejaoturi lahendus ei ole vajalik. Antud variant võib-olla kiirem, kuigi *Deployment* ja *StatefulSet* kontrollerid ei ole täiesti identsete võimalustega. Lisaks sellest, et tagada sessiooni püsivust on võimalik kasutada *Nginx* konfiguratsiooni *ip_hash*, mis peab garanteerima seda, et samast *IP* aadressist päringud jõuavad samale rakenduse eksemplarile. Aga *CRM* valdkonna rakendused on kõik *Stateless* ehk kõik päringud hoiavad sessiooni info endas ning sellised konfiguratsioonid ei ole vajalikud.

Alternatiivse täiendava lahendusena võib vaadelda kontseptsiooni, mida pakuvad D. Ernst, A. Becker ja S. Tai oma artiklis [30], kus nad pööravad tähelepanu sellisele probleemile nagu ressursside kasutus kanaarjuurutuse raames. Artikli autorid väidavad, et kanaarjuurutus koormab infrastruktuuri ning kanaarjuurutust on raske monitoorida kuna kanaari ja stabiilse rakenduse versioonid saavad erinevat võrguliikluse koormust. Nende väitega võib nõustuda, kuna tõe poolest kanaarjuurutus loob rakenduse lisa eksemplare ning tingimused infrastruktuuris, kuhu on paigaldatud kanaarversioon, ei ole identsed võrreldes infrastruuriga, kuhu on paigaldatud ainult stabiilne versioon. Artikli autorid pakuvad juurutada kanaari ja stabiilset versiooni samadesse tingimustesse ning kasutada selle jaoks lisa koormusejaoturit, mis jaotaks ressursse võrdselt ja monitooriks kõikide päringute infot. Lisaks artiklis pakutakse monitoorida ka hoiatuste tasemel (ingl k *warning*). Lõputöö autor võib nõustuda antud artiklis püstitatud probleemidega ja arvestab pakutud lahendustega tulevastes jätkutöödes.

3.2.3 Järeldused seoses kanaarväljalaskega

Kanaarväljalaske realisatsioon annab head võimalust hakata praktiseerima paindlikumaid väljalaske strateegiaid olukordades kus see tundub vajalikuna. Antud strateegia vähendab vigade arvu toodangu keskkonnas (*KPI*: *Live Critical Error Count*), ning aitab hoida arendustiimi kindlust rakenduse kvaliteedis. Kanaarväljalaske puhul selline *KPI* nagu arenduse aeg (*Dev time*) muidugi võib minna ülesse, kuid mitte väga palju, kuna uus versioon oleks kanaari staatuses ainult kaks tundi ning kanaari aeg võib minna toodangu testimise alla.

Autor teeb kanaarväljalaske staadiumi pidevjuurutamise protsessi püsivaks osaks koos automaatse tagasikeeramise võimalusega, millest käib jutt peatükis 3.4 „*Plumbr-i* vigade taseme alusel versiooni automaatse tagasipöörde teostamine *Bamboo* platvormil“

3.3 Monitooring

Monitooringu täiendamine seisneb selles, et *Reservation* rakendus oli täiendatud *Prometheus* meetrikatega. *Zabbix* platvormil *Reservation* hostile oli lisatud integratsioon *Plumbr Reservation* monitooringuga ning lisatud triger, mille ülesanne on panna püsti alarm *Juhtimiskeskusele* juhul kui R*eservation* vigade tase on kõrgem kui 3% miinimum 3 korda, küsides seda iga 10 minuti tagant.

3.3.1 Prometheus monitooringu lisamine

Prometheus on monitooringu süsteem koos mälusisese dimensioonilise aegridade andmebaasiga. Antud süsteem laseb korjata väga suure hulga meetrikaid, kasutab neid graafikute genereerimiseks ning saadab *alerte* juhul kui meetrikad ületavad määratud limiite. *Prometheus* kasutab *PromQL* (***P****rometheus* ***Q****uery* ***L****anguage*) selleks, et pärida ja agregeerida andmeid aegridade formaadis reaalajas. *Prometheus* süsteemis eksisteerib oma visualiseerimise liides graafikute loomiseks, kuid tihedamini kasutatakse selleks teisi süsteeme, nagu *Grafana* [32]. Integratsioon *Prometheus* monitooringuga annab võimalust jälgida ka ärifunktsionaalsuse meetrikaid.

Lahenduse algoritm sai olema järgmine:

- *Spring Boot 2.2* versiooni sisseehitatud *Spring Boot Actuator* [31] ja *Micrometer* [33] korjavad *Kubernetes Pod*-idest meetrikaid ja võrguliiklust;
- saadud meetrikaid saab vaadata *Prometheuse* kasutajaliideses ja kasutada integreeritud süsteemides.

Selleks, et hiljem luua graafikuid lisa meetrikate alusel ja tagada ülemineku võimalust Prometheus monitooringule autor määras ära meetrikad, mida ta peab oluliseks *Reservation* rakenduse jaoks.

Need meetrikad koos päringutega on kajastatud alloleval koodilõigul:

- *Reservation* rakenduse üldine vigade tase;

- *Reservation* rakenduse HTTP 500, 404, 403, 400 vigade agregatsioon iga 10 minuti jooksul;
- teenuse put /reservation/{reservation_id}/cancel „Cancel reservation“ – teenuse vastuse kiirus ja väljakutsumiste arv;
- teenuse post /reservation/branches/{branch_id} „Create reservation to branch“, – teenuse vastuse kiirus ja väljakutsumiste arv;
- teenuse get /reservation/available/branches/{branch_id} „Get available times by branch“ – teenuse vastuse kiirus ja väljakutsumiste arv;
- teenuse get /branches „Get all branches with type STORE and TECH_STORE“ – teenuse vastuse kiirus ja väljakutsumiste arv;
- teenuse get /availbale-slots-by-branches „Find available slots by branches with type STORE or TECH_STORE“ – teenuse vastuse kiirus ja väljakutsumiste arv;
- 99. protsentiil;
- Tootlikust (ingl k *Throughtput*).

Allolev koodilõik [Joonis 16] näitab kuidas kasutades *PromQL'i* ja olemasolevaid Prometheus näitajatest võib saada eelnevalt defineeritud meetrikaid:

```
#Reservation rakenduse HTTP 500, 404, 403, 400 vigade agregatsioon iga 1 minuti jooksul;
sum(rate(http_responses_total{code="XXX"}[1m])) / sum(rate(http_responses_total[1m]))

#Reservation rakenduse HTTP 500, 404, 403, 400 vigade agregatsioon nädala lõikes;
sum(http_responses_total{code="XXX"} offset 1w)
#teenuse put /reservation/{reservation_id}/cancel „Cancel reservation“ – teenuse
vastuse kiirus ja väljakutsumiste arv;

Counter:
http_requests_total{container="reservation-spring", handler= ~".*/cancel"
}[1m]

Timer:
rate(http_response_time_sum{container="reservation-spring",handler=~".*/cancel", status="200"}[1m])
/
rate(http_response_time_count{container="reservation-spring",handler=~".*/cancel", status="200"}[1m])
```

Joonis 16. Prometheus PromQL.

PromQL päringut tervikuna võib leida plokis Lisa 2 „*Prometheus PromQL*“

Lisaks lähtudes neljast kuldsest signaalist [18], kus iga monitooringu raames on vaja kindlasti jälgida vigade taset, võrguliiklust, latentsust ja küllastumist (ingl k *Saturation*) autor lisas ka neid meetrikad *Reservation* rakenduse monitooringusse.

- Võrguliiklus
- Latentsus
- Küllastumine (ingl k *Saturation*)

Joonis [Joonis 17] näitab tehtud päringuid lähtudes monitooringu neljast kuldsest signaalist:

```
#Reservation rakenduse üldine vigade tase;
errors_total[1m] - Integratsioonis grok_exporter-iga

#Võrguliiklus
sum(rate(http_requests_total[1m]))

#Latentsus
sum(rate(http_request_duration_seconds_bucket{le="0.4"}[1m]))
/ sum(rate(http_request_duration_seconds_count[1m]))

#Küllastumine (ingl k Saturation)
avg_over_time(cpu[1m])
avg_over_time (node_disk_io_time_seconds_total[1m])
```

Joonis 17. Prometheus meetrikad ja neli kuldset signaali.

Kaanarjuurutuse kontekstis on vaja filtreerida *container* parameetri alusel, mis kanaarjuurutuse puhul on *„reservation-spring-canary“*.

Lisaks selleks, et saaks jälgida detailsemaid meetrikaid teenuste tasemel autor tegi integratsiooni *Micrometer* raamistikuga, see annab võimalust kontrollida mis hetkel mõõtmine algab ja lõpeb. Allolev koodilõik [Joonis 18] näitab kuidas oli teostatud teenuse /branches vastuse kiiruse kontroll:

```
private final CollectorRegistry collectorRegistry;
Histogram getBranchDuration;
Counter fetchFail;
@PostConstruct public void init() {
getBranchDuration = Histogram.build()
.name("get_branch_duration")
.help("Time elapsed for get branch duration").register(collectorRegistry);

fetchFail = Counter.build()
.name("close_reservation_admin")
.help("Fail counter - closing reservations from admin")
.register(collectorRegistry);
}
public List getExistingByBranchInDateRange(Integer branchId, LocalDateTime from,
LocalDateTime to) {
Histogram.Timer getTimer = getBranchDuration.startTimer();
List branchIds = branchAdminService.findBranchAndChildBranchIds(branchId);
List dateReservations = reservationRepository.findUpcomingByBranchIds(branchIds,
from.toLocalDate(), to.toLocalDate())
.stream()
.filter(reservation -> RESERVED.equals(reservation.getStateCode())) .collect(toList());
double getElapsed = getTimer.observeDuration(); // Writes to actuator <-- Time elapsed
return filterReservationsFor(dateReservations, from, to);
}
```

Joonis 18. Detailne meetrika.

Üldine vigade tase ning konkreetsete vigade kasv ja latentsus on need meetrikad, mis määravad ära kas kanaariväljalase vastab kvaliteetide nõudmistele. Vigade arv, latentsus, võrguliiklus ja küllastumine on põhiindikaatorid, mis näitavad 24/7 rakenduse seisukorda.

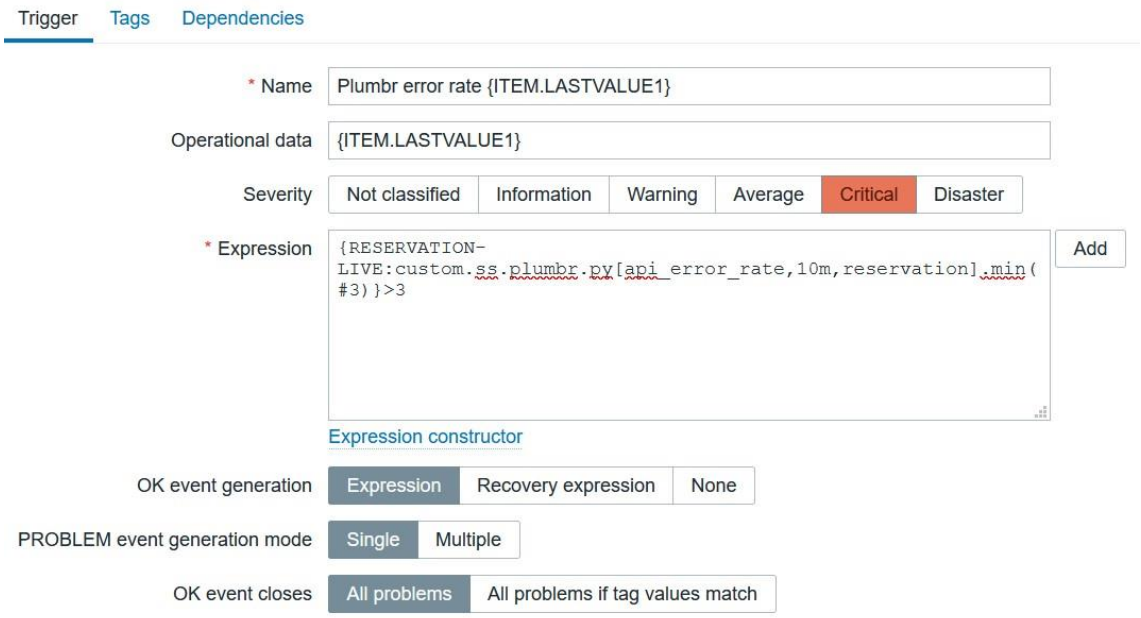
3.3.2 *Reservation* rakenduse vigade taseme alusel kriitiliste alertide loomine Zabbixi platformil

Zabbix on platform,mis teostab monitooringut võrgu, serverite, teenuste, pilveressursside ning ärinäidete tasemel. *Reservation* rakenduse kontekstis kasutatakse praegu *Zabbix* süsteemi infrastruktuuri seisu jälgimiseks. Antud töös realiseeritakse integratsioon *Zabbix* ja *Plumbr* süsteemide vahel.

Lahenduse algoritm sai järgmine:

- lisada *Reservation* hostile integratsioon *Plumbr* süsteemiga, mis päriks viimase käest vigade taset iga 10 minuti tagant;
- lisada triger, mis vaataks kas kolme viimase tulemuse väärtus oli võrdne või suurem kui 3 %;
- juhul kui vigade tase oli kõrgem kui 3%, siis *Zabbix* loob kriitilise *alarm*-i, mida jälgib *Juhtimiskeskus*:
- *Juhtimiskeskus* helistab rakenduse administraatoritele;
- juhul kui vigade tase on madalam kui 3% jätkata meetrika jälgimist.

Allolev joonis [Joonis 19] näitab kuidas näeb välja *Zabbix* kriitilise alarmi konfigureerimine, kasutades trigerit:



Joonis 19. Zabbix alarmi konfiguratsioon.

Lisainformatsioon asub antud lõputöö plokis Lisa 3 „*Zabbix* alarmi konfigureerimine“

3.3.3 Monitooringu alternatiivsed lahendused

Alternatiivse lähenemisena võib välja tuua *Plumbr* süsteemi täiendamist, tellides sealt uusi funktsionaalsusi: alarmid *Juhtimiskeskusele*, lisa graafikuid.

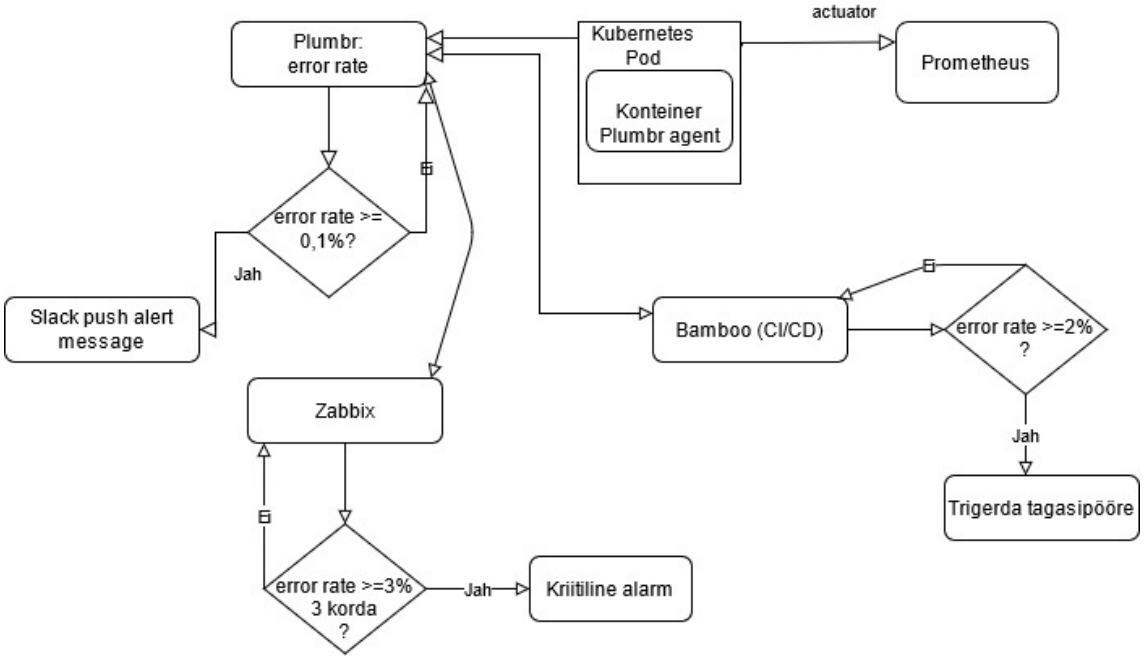
Lisaks võib uurida kas *Zabbix* süsteemis on olemas hea visualiseerimise kiht. Kuigi see igal juhul eeldab *Prometheuse* kasutamist vahekihina.

Kuna *Plumbr* lõpetab oma iseseisvat tööd ja ühineb teise monitooringu tootega – *Splunk* võib uurida võimalust täiesti üle viia meetrikad (mida praegu jälgitakse *Plumbr* poolt) *Prometheus* monitooringu andmebaasi ja sealt edasi luua kõiki vajalikke graafikuid *Grafanas.* Ideaalis kõik vajalik informatsioon võiks olla konsolideeritud ühes monitooringu süsteemis, mis teostab kõiki vajalikke tegevusi ja sobib organisatsioonile.

3.3.4 Järeldused seoses monitooringuga

Monitooringu protsessi täiendamises oli tehtud integratsioon *Prometheuse* monitooringu andmebaasiga, püstitatud nõudmised jälgitavate meetrikate suhtes ning konfigureeritud kriitilised alarmid *Zabbix* süsteemis.

Allolev joonis [Joonis 20] näitab uuendatud monitooringu protsessi:



Joonis 20. Pidevmonitooring.

Antud uuendus tagab pidevat jälgimist ja reageerimist *Reservation* rakenduse monitooringus ning sellest võib saada monitooringu protsessi muster, mida edaspidi võib täiendada lähtudes vajadustest.

3.4 *Reservation* rakenduse vigade taseme alusel versiooni automaatse tagasipöörde teostamine *Bamboo* platvormil

Antud realisatsioon oli tehtud selleks, et kasutada koos nii kanaari kui ka stabiilse versioonide väljalaskmisega.

Realisatsioon seisneb selles, et pidevintegratsiooni ja pidevvalmiduse süsteemi lisada *Reservation* rakenduse *Deployment* projekti *bash* skript, kasutades *cURL* süntaksit.

Autor lõi kaks lahenduse algoritmi: esimene teeb versioonile tagasipööret kohe pärast esimest vigade taseme tõusmist, teine – 3 viimaste väärtuse alusel.

Esimene lahenduse algoritm on järgmine:

1. Skript alustab tööd 10 minuti pärast uue versiooni juurutust
2. skript teeb autentimise tehnilise kasutajaga *Plumbr REST API*[1] pihta *GET* meetodiga, mis annab tagasi vigade taset viimase 10 minuti jooksul;
3. juhul vigade tase on suurem kui 2%, kutsub välja pidevintegratsiooni ja -juurutuse süsteemi trigeri [35] mis pärib viimaste versioonide väljalaske nimekirja ja kutsub välja juurutuse protsessi eelviimase versiooni väljalaske ID-ga

Teine lahenduse algoritm on järgmine:

1. Skript alustab tööd 10 minuti pärast uue versiooni juurutust
2. skript teeb autentimise tehnilise kasutajaga *Plumbr REST API* pihta *GET* meetodiga, mis annab tagasi vigade taset viimase 10 minuti jooksul;
4. salvestab saadud tulemuse massiivi ERROR_RATE;
5. ootab 10 minutit, kasutades pidevintegratsiooni ja pidevvalmiduse süsteemi *Scheduler* [36]
6. kordab sammud 1-3 veel kaks korda (salvestades kokku 3 väärtust)
7. itereerib massiivi ERROR_RATE, juhul kui kõik 3 väärtust massiivis ERROR_RATE on suuremad kui 2%, kutsub välja pidevintegratsiooni ja -juurutuse süsteemi trigeri [35], mis pärib viimaste versioonide väljalaske nimekirja ja kutsub välja juurutuse protsessi eelviimase versiooni väljalaske ID-ga
8. ERROR_RATE massiivi tühjendamine

[1] Integratsiooni arhitektuuriline stiil komponentide vahel hajutatud süsteemis

Lisainformatsioon skripti kohta asub plokis „Lisa 4 – Automaatne tagasipööre eelmisele versioonile (1 väärtus)“ ja Lisa 5 – „Automaatne tagasipööre eelmisele versioonile (3 väärtust)“.

3.4.1 Versioonide tagasipöörde alternatiivsed lahendused

Versioonide tagasipöörde alternatiivseteks lahendusteks võib nimetada selliseid pidevintegratsiooni, -valmiduse, -juurutuse süsteeme, kus eksisteerivad võimalused teostada kanaarjuurutust koos monitooringu süsteemi integratsiooniga ja probleemide korral teha automaatset tagasipööret. Nende süsteemide hulgas on *Gitlab CI* [25], *Spinnaker+Consul* [37] või *Spinnaker+Istio* [28], kus on juba integreeritud tugi automaatse kanaarjuurutuse jaoks. Juurutuse konveieril saab kanaari väljalase esialgu, näiteks, 5% võrguliiklusest. Juhul kui *Prometheus* ei fikseeri probleeme, lähtudes meetrikate näitajatest, kasvab võrguliiklus 50%-le. Juhul kui ka pärast seda *Prometheus* ei leia probleeme, muutub kanaarversioon primaarseks versiooniks. Juhul kui kanaarväljalaskega esinesid probleemid teeb Spinnaker tagasipööret eelmisele versioonile. Võrguliikluse suurendamise etappe võib veel lisada ka 10% või 20% sammuga: 10% -> 30% -> 50%-> 70%.

Muidugi on palju mugavam, kui kõik vajalikud vahendid on kohe olemas samas süsteemis ja perioodiliselt IT osakonnad peavad uuendama oma tehniliste vahendite komplekti. Autori arvamusel, ajal kui paljud firmad juba rakendavad pidevjuurutuse protsessi, tasub loobuda vahenditest, mis kunagi olid mõeldud põhimõtteliselt ainult pidevjuurutuse jaoks.

3.4.2 Järeldused versioonide tagasikeeramisest

Tagasipöörete kontseptsioon on üsna levinud praktika, mida kasutatakse enamasti kanaarjuurutusega. Kuigi autori arvamusel seda võib rakendada ka pärast primaarse versiooni juurutust kahetunnise perioodi jooksul.

Antud kontseptsiooni realiseerimises tekkisid ka küsimused:

- Kas eelmine versioon garanteerib probleemse olukorra eemaldamist? Lahendusena siin võib praegu välja tuua seda, et versiooni vahetusi tehakse esmaspäevast kuni neljapäevani kellavahemikus 9.00-16.00 ja probleemsest olukorrast kindlasti saab teada ka valveisik

- Kas tagasikeeramise protsess võib minna tsüklisse, kus hakatakse vahetama kaks viimast versiooni? Lahendusena võib pakkuda veel hoida tagasikeeratud versioonide nimekirja.

Autori arvamusel automaatne tagasipööre eelmisele versioonile probleemse kanaarjuurutuse raames on õige lähenemine, mida kasutamise ajal saab alati redakteerida.

Edasine lähenemine siin võib seisneda ka selles, et rakendada automaatseid versioonide tagasikeeramisi koos enesediagnostika süsteemiga, mis oskaks võrrelda sama jälgitava teenuse vastuseid ja näitajaid eelmistes versioonides. Väljalaskmiseks valiks selle versiooni, kus vastused ja näitajad olid õiged.

4 Järeldused

Teoreetilise materjali analüüs ning läbiviidud küsitlus näitasid, et olemasolevad pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsessid vajasid täiendamist selleks, et tagada veelgi paremat kvaliteedi kontrolli *Reservation* rakenduses ning parendada arenduse *KPI* väärtused. Protsesside täienduse ettepanekud olid formuleeritud läbiviidud küsitluse tulemuste alusel, kust oli näha tegevusi testimisest, versioonide väljalaskest ja monitooringust. Respondentide arvamusel need omavad kõige rohkem mõtet ja on kulutatud aega väärt. Vanad ja uuendatud protsessid olid võrreldud ning olid välja toodud nende erinevused. Protsessi täienduse realisatsiooni võib kasutada ka teistes Telia *CRM* rakendustes. Perspektiivis rakenduse versioonide juurutamise ning tagasipöörde koodi võib välja viia eraldi koodihoidlasse, kust seda saavad lihtsalt taaskasutada teised projektid. Antud lähenemine aitaks paremini ühtlustada projektide pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringuprotsesse. Mõju mida uuendatud protsessid toovad endaga kaasa võib illustreerida alloleva tabeliga [Tabel 7]:

Tabel 7. Vana ja uue protsesside võrdlus ning mõju KPI peale.

Protsessi aspekt	**Tänane**	**Uus**	**Mõju (*KPI*)**
Rakenduse regressiooni seis	Kasutajaliidese integratsiooni automaattestid puuduvad	Kasutajaliidese integratsiooni testid käivituvad pidevintegratsiooni serveril iga *build* protsessiga ja igas harus	Toodangu vigade arv langeb ↓ Arenduse aeg väheneb ↓ Aeg uue arenduse väljalaskmiseni ↓
Rakenduse uue versiooni järkjärguline väljalase	Tänases protsessis puudub võimalus teha rakenduse uus versioon kättesaadavaks ainult osalisele protsendile klientidest, selleks et minimiseerida potentsiaalsetest vigadest mõjutatud kasutajate arvu.	Uues protsessis ilmub võimalus teostada kanaarjuurutust, mille käigus ainult teatud protsent kasutajatest saab ligi rakenduse uuele versioonile	Toodangu vigade arv langeb ↓ Arenduse aeg väheneb ↓ Aeg uue arenduse väljalaskmiseni ↓

Rakenduse lisa meetrikate korjamine	Tänases rakenduses puudub võimalus korjata *Kubernetes Pod*-idest paindlikumaid meetrikaid	Uues protsessis *Prometheus* funktsionaalsus hakkab korjama lisa meetrikad *Kubernetes Pod*-idest ja looma nende alusel informatiivseid graafikuid	Toodangu vigade arv langeb rakenduse täpsema analüüsi kaudu↓ Kriitiliste vigade lahendamise keskmine aeg langeb ↓ Aeg uue arenduse väljalaskmiseni↓
Rakenduse infoteated vigade taseme tõusemisest kriitilisele tasemele töövälisel ajal (pühad, õhtune aeg ja nädalavahetused)	Praegu infoteated tulevad ainult arendustiimi *Slack* kanalisse, mida valvel olev arendustiimi liige jälgib ainult tööajal	Uues protsessis monitooringu süsteem *Zabbix* hakkab pärima rakenduse vigade taset ja selle alusel teeb järeldusi ning vajaduse korral loob *alarmi Juhtimiskeskusele*	Toodangu vigade vähendamine ↓ Kriitiliste vigade lahendamise keskmine aeg langeb ↓
Rakenduse versiooni automaatne *tagasipööre*	Praegu rakenduse versioonide *tagasipööre* vajab alati inimese sekkumist isegi kui vigade tõttu seis on kriitiline	Uues protsessis ilmub võimalus rakenduse kõrge vigade taseme juhul teostada automaatne tagasipööre *CI/CD* süsteemi poolt	Toodangu vigade vähendamine ↓

CI – pidevintegratsioon, *CD* – pidevvalmidus/-juurutus

Eraldi peatükis 5. on väljatoodud edasised plaanid, mis tekkisid lõputöös tehtud analüüsi ja realisatsiooni raames.

Numbrites *KPI* väärtuste planeeritud muudatused on järgmised:

- arenduse aeg (ingl k *Story development time*): 9,3 päeva -> 8,5 päeva;
- aeg väljalaskmiseni (ingl k Time To Market): 44,7 päeva -> 40,0 päeva
- toodangu keskkonna kriitliste ja oluliste vigade keskmine lahendamise aeg (ingl k *Critical and Major Live bugs resolution average time*): 0,07 päeva (1 tund 41 minutit) -> 0,05 päeva (1 tund 12 minutit);
- toodangu keskkonna vigade arv Sprindis (ingl k *Live bugs count*): 8 tk -> 5 tk.

Arenduse aja näitaja väheneb juba tänu sellele, et mahukate muudatuste järel on testimise etapp nüüd kiirem. Ilma regressiooni automaattestideta manuaalne regressioon oli võtnud kaks tundi aega, nüüd seda teostatakse automaatselt 15 minutiga.

Teiste *KPI* näitajate muudatus võtab aega, aga moodustub vähemalt poole kuu pärast.

Kõik ülesanded, mis oli püstitatud peatükis 1.2 on täidetud ning eesmärk on saavutatud.

5 Edasised plaanid

Analüüs ja läbiviidud küsitlus näitasid, et protsesse oleks mõistlik täiendada veel teistega automaattestide liikidega nagu koormuse (ingl k *Performance tests*), turva (ingl k *Penetration* tests), võrdluse (ingl k *Tap-comparison tests)* ja toodangu testid (ingl k *Smoke tests*). Lisaks küsitlusele vastajad toetasid automaatsete infoteadete saatmist versiooni väljalaske kohta, mis annaks kõikidele huvitatud osapooltele paremat ülevaadet toodangu seisu kohta. Positiivse hinnangu said ka ideed paindlikuma monitooringu kohta, mida pakub praegu *Prometheus* integratsioonis *Grafana*. Võimalik, et tulevikus olemasolevad monitooringu süsteemid võiksid üle võtta monitooringut teistest süsteemidest, mis toetaks küsitlusest saadud seisukohta, et üks ainus monitooringu süsteem kiirendaks probleemi uurimist. Monitooringu süsteemide enesediagnostika ja eneseparandus näeb kindlasti välja perspektiivsena ja on uurimist väärt. Enesediagnostika ja eneseparandamise printsiipi võib kasutada nii süsteemides kui ka konkreetsete skriptide ja teenuste tasemel. Saadud tagasisidest autor loob jätkutööd, mida tema ja tema arendustiim saavad analüüsida ja realiseerida innovatsiooni Sprindi käigus [38]. Vaja eraldi mainida, et rakenduste versioonide väljalaske, monitooringu ning versioonide tagasipöörde funktsionaalsuste realiseerimine võib minna hoopis teisele tasemele, juhul kui kasutusel on platvormid, mis juba kohe alguses omavad läbimõeldud tuge antud funktsionaalsustele. Uute platvormide juurutamine kindlasti ei ole lihtne ega odav töö. Paremini argumenteerida vajadust uutes platvormides, saab selliste realisatsioonide juurutamise näitel nagu olid tehtud antud lõputöös.

6 Kokkuvõte

Lõputöö eesmärgiks oli parendada *Reservation* rakenduse arenduse *KPI* väärtused. Arenduse *KPI* väärtused on oluline osa arendusvaldkonna aasta hindamiseks, seega nende paremaks muutmine on väga aktuaalne. Püstitatud hüpotees seisnes selles, et antud eesmärki võib saavutada pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside täiendamise kaudu. Tegevused seas, mis olid vajalikud selle jaoks, oli läbi analüüsida praegust pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesse, määrates nendest tegevusi, mis tagavad efektiivsust ja kvaliteeti ja neid, mis vajavad täiendamist. Arendustiimile ühe hiljuti üleantud rakenduse alusel autor määras ära missugused osad protsessidest peavad olema lisatud , seletades mida tagab iga nendest muudatusest.

Antud lõputöö raames autor ka viis läbi küsitluse selleks, et teada saada arendajate, arhitektide, administraatorite ning juhtide arvamust efektiivse pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu protsesside kohta. Küsitluse vastuste põhjal oli tehtud planeeritud tööde prioriseerimine.

Lähtudes tehtud analüüsist, lõputöö raames autor teostas ära ka realisatsiooni, mis hõlmab järgmisi pidevintegratsiooni, -valmiduse, -juurutuse ning -monitooringu plokke nagu automaattestimine, väljalaske strateegia, monitooringu konfiguratsioon ja tagasipööre eelmisele versioonile.

Testimises autor realiseeris komplekti kasutajaliidese integratsiooni teste rakenduse regressiooni otstarbeks. Väljalaske strateegia plokis lõi võimaluse teostada kanaarjuurutamist. Monitooringu konfiguratsiooni plokis olid sätestatud meetrikad rakenduse täpsemaks monitooringuks. Lisaks autor lisas monitooringu töövälisel ajal ja uue monitooringu süsteemi kasutamise, mis annab paremat ülevaadet rakenduse seisust ning mille kaudu võib konfigureerida paindlikumaid graafikuid ning tulevikus ka info -ja veateadete väljasaatmist. Loogilises komplektis kanaarjuurutamisega ja monitooringuga oli realiseeritud võimalus teostada automaatseid tagasipöördeid rakenduse eelmisele versioonile.

Tehtud analüüs ja realisatsioon omab praktilist väärtust Telia Eesti AS jaoks. Realisatsioon on rakendatud *Reservation* rakendusse ning analüütiline tulemus aitab optimeerida pidevintegratsiooni, -valmiduse, -juurutuse ja -monitooringu protsesse tulevikus.

Realisatsioonist saadud mustreid võib edaspidi rakendada ka muudes rakendustes, kus kohe ei olnud võimalust teostada eksperimente nende rakenduste väga suurte hulga tarbijate tõttu.

Kasutatud kirjandus

[1] Atlassian ‚*SLA vs. SLO vs. SLI: What's the difference*?, 2018. [Võrgumaterjal]. Loetud aadressil: https://www.atlassian.com/incident-management/kpis/sla-vs-slo-vs-sli. Kasutatud: 22.04.2021.

[2] M. Manturewicz, „What is CI/CD - all you need to know," 17 April 2019. [Võrgumaterjal]. Loetud aadressil: https://www.codilime.com/what-is-ci-cd-all-you-need-to-know/. Kasutatud: 18.04.2021.

[3] Software Testing Fundamentals, *Unit Testing*, 13 September 2020. [Võrgumaterjal]. Loetud aadressil: https://softwaretestingfundamentals.com/unit-testing/. Kasutatud: 16.04.2021.

[4] Software Testing Fundamentals, *Acceptance Testing*, 13 September 2020. [Võrgumaterjal]. Loetud aadressil: https://softwaretestingfundamentals.com/acceptance-testing/. Kasutatud: 10.04.2021.

[5] Software Testing Help, *What Is Regression Testing? Definition, Tools, Method, And Example*, 30 March 2021. [Võrgumaterjal]. Loetud aadressil: https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/. Kasutatud: 19.04.2021.

[6] S. Korolev, „T-shaped Skills in Product Development," 25 May 2018. [Võrgumaterjal]. Loetud aadressil: https://railsware.com/blog/t-shaped-skills-in-product-development/. Kasutatud: 18.04.2021.

[7] G. Sypolt, „The Why and How of Tap Compare Testing," 7 August 2018. [Võrgumaterjal]. Loetud aadressil: https://saucelabs.com/blog/the-why-and-how-of-tap-compare-testing. Kasutatud: 15.04.2021.

[8] Stackify, *The Ultimate Guide to Performance Testing and Software Testing: Testing Types, Performance Testing Steps, Best Practices, and More*, 16 April 2021. [Võrgumaterjal]. Loetud aadressil: https://stackify.com/ultimate-guide-performance-testing-and-software-testing/. Kasutatud: 30.04.2021.

[9] Cloudflare, *What Is Penetration Testing? What Is Pen Testing?*, 2021. [Võrgumaterjal]. Loetud aadressil: https://www.cloudflare.com/learning/security/glossary/what-is-penetration-testing/. Kasutatud: 30.04.2021.

[10] Software Testing Fundamentals, *Smoke Testing*, 2 September 2020. [Võrgumaterjal]. Loetud aadressil: https://softwaretestingfundamentals.com/smoke-testing/. Kasutatud: 20.04.2021.

[11] S. Pittet, "Continuous integration vs. continuous delivery vs. continuous deployment," Atlassian, 2021. [Võrgumaterjal]. Loetud aadressil: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. Kasutatud: 18.04. 2021.

[12] J. Humble and D. Farley, *Continuous delivery vs continuous deployment*, Addison-Wesley, 2010.

[13] LauchDarkly,*Deployment and release strategies*, 21 May 2020. [Võrgumaterjal]. Loetud aadressil: https://docs.launchdarkly.com/guides/best-practices/deployment-strategies. Kasutatud: 18.04.2021.

[14] Itay Shakury, „Deployment Strategies Defined,“ 20 November 2017. [Võrgumaterjal]. Loetud aadressil: http://blog.itaysk.com/2017/11/20/deployment-strategies-defined. Kasutatud: 19.04.2021.

[15] J. Hirschauer, „Continuous Deployment and Continuous Monitoring: A Winning Pair,“ 16 June 2018. [Võrgumaterjal]. Loetud aadressil: https://dzone.com/articles/continuous-deployment-and-continuous-monitoring-a. Kasutatud: 18.04.2021.

[16] Sumologic,*Continuous Monitoring*, 2018. [Võrgumaterjal]. Loetud aadressil: https://www.sumologic.com/glossary/continuous-monitoring/. Kasutatud: 19.04.2021.

[17] C. Kidd, „Tracing vs Logging vs Monitoring: What’s the Difference?,“ 7 March 2019. [Võrgumaterjal]. Loetud aadressil: https://www.bmc.com/blogs/monitoring-logging-tracing/. Kasutatud: 19.04.2021.

[18] R. Ewaschuk, „ The Four Golden Signals*”* in *Site Reliability Engineering: How Google Runs Production Systems*, O’Reilly, 2016. [Võrgumaterjal]. Loetud aadressil: https://sre.google/sre-book/monitoring-distributed-systems/#xref_monitoring_golden-signals. Kasutatud: 30.04.2021.

[19] C. Jones, J. Wilkes, N. Murphy and C. Smith, „ Service Level Objectives*”* in *Site Reliability Engineering: How Google Runs Production Systems*, O’Reilly, 2016. [Võrgumaterjal]. Loetud aadressil: https://sre.google/sre-book/service-level-objectives/. Kasutatud: 30.04.2021.

[20] G. Schermann, J. Cito and P. Leitner, “Continuous Experimentation: Challenges, Implementation Techniques, and Current Research,” IEEE Software, vol. 35, pp. 26 - 31, 2018, doi: 10.1109/MS.2018.111094748. Kasutatud: 18.04.2021.

[21] Test guild, *4 Top Automation Testing Design Patterns (Plus 86 More)*, 2018. [Võrgumaterjal]. Loetud aadressil: https://testguild.com/automation-testing-design-patterns/. Kasutatud: 17.04.2021.

[22] T. Rhodes, „ Evaluating Cypress and TestCafe for end to end testing,“ 29 March 2018. [Võrgumaterjal]. Loetud aadressil: https://medium.com/yld-blog/evaluating-cypress-and-testcafe-for-end-to-end-testing-fcd0303d2103. Kasutatud: 19.04.2021.

[23] Marko Lukša, *Kubernetes in Action*, Manning (Shelter Island), 2018.

[24] NGINX, *Module ngx_http_upstream_module*, 2010. [Võrgumaterjal]. Loetud aadressil: http://nginx.org/en/docs/http/ngx_http_upstream_module.html. Kasutatud: 17.04.2021.

[25] K. Wuestkamp, „Kubernetes Canary Deployment #1 Gitlab CI,“ 5 January 2020. [Võrgumaterjal]. Loetud aadressil: https://levelup.gitconnected.com/kubernetes-canary-deployment-1-gitlab-ci-518f9fdaa7ed. Kasutatud: 2.05.2021.

[26] K. Wuestkamp, „Kubernetes Canary Deployment #2 Argo Rollouts,“ 13 January 2020. [Võrgumaterjal]. Loetud aadressil: https://codeburst.io/kubernetes-canary-deployment-2-argo-rollouts-5e68e99b4fa3. Kasutatud: 1.05.2021.

[27] K. Wuestkamp, „Jenkins-X Istio Flagger Canary Deployment,“ 24 February 2020. [Võrgumaterjal]. Loetud aadressil: https://itnext.io/jenkins-x-istio-flagger-canary-deployment-9d5e187c2334. Kasutatud: 1.05.2021.

[28] K. Wuestkamp, „Kubernetes Istio Canary Deployment,“ 3 February 2020. [Võrgumaterjal]. Loetud aadressil: https://itnext.io/kubernetes-istio-canary-deployment-5ecfd7920e1c?gi=ff32df353839. Kasutatud: 1.05.2021.

[29] R. Ducott, „Two-phased Canary Rollout with Open Source Gloo,“ 22 April 2020. [Võrgumaterjal]. Loetud aadressil: https://kubernetes.io/blog/2020/04/two-phased-canary-rollout-with-gloo/. Kasutatud: 15.03.2021.

[30] D. Ernst; A. Becker and S. Tai, “Rapid Canary Assessment Through Proxying and Two-Stage Load Balancing,” in Proceedings of 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, 2019. Loetud aadressil: https://ieeexplore.ieee.org/abstract/document/8712149. Kasutatud: 18.04.2021.

[31] J. C. V. Sánchez, „Spring Boot Actuator,“ 30 January 2021. [Võrgumaterjal]. Loetud aadressil: https://www.baeldung.com/spring-boot-actuators. Kasutatud 20.04.2021.

[32] D. Singh, „Monitoring Spring Boot Apps with Micrometer, Prometheus, and Grafana,“ 1 May 2020. [Võrgumaterjal]. Loetud aadressil: https://stackabuse.com/monitoring-spring-boot-apps-with-micrometer-prometheus-and-grafana/. Kasutatud 20.04.2021.

[33] Micrometer, *Micrometer Prometheus*, 2020. [Võrgumaterjal]. Available: https://micrometer.io/docs/registry/prometheus. Kasutatud 20.04.2021.

[34] Prometheus, *What is Prometheus?*, 2014. [Võrgumaterjal]. Loetud aadressil: https://prometheus.io/docs/introduction/overview/. Kasutatud 25.04.2021.

[35] Atlassian, *REST API deployment triggers for Bamboo*, 2017. [Võrgumaterjal]. Loetud aadressil: https://developer.atlassian.com/server/bamboo/rest-api-deployment-triggers-for-bamboo/. Kasutatud: 27.04.2021.

[36] Atlassian, *Cron-based scheduling*, 2014. [Võrgumaterjal]. Loetud aadressil: https://confluence.atlassian.com/bamboo/cron-based-scheduling-289276907.html. Kasutatud: 27.04.2021.

[37] R. Wang, „ Automated Canary Deployment with HashiCorp Consul and Spinnaker,“ 22 April 2021. [Võrgumaterjal]. Loetud aadressil: https://www.hashicorp.com/blog/automated-canary-deployment-with-hashicorp-consul-and-spinnaker/. Kasutatud: 7.05.2021.

[38] Scaled Agile, *Innovation and Planning Iteration*, 2021. [Võrgumaterjal]. Loetud aadressil: https://www.scaledagileframework.com/innovation-and-planning-iteration/. Kasutatud: 6.05.2021.

Lisa 1 – Testcafe kasutajaliidese testid

```
import '../conf/process-env.js';
import ReservationTestHelper from '../utils/ReservationTestHelper';

fixture `Changing working hours for the STORE`
  .page (process.env.UI_TEST_ENV_URL + '#/branches')
  .httpAuth({
    username: process.env.DEALER_USERNAME,
    password: process.env.DEALER_PASSWORD,
    domain:      'ET.EE'
  })
  .afterEach(async (t) => {
    const { error } = await t.getBrowserConsoleMessages();
    console.log(error);
  });

test ('Dealer can change working hours for regular time', async () =>  {
  await new ReservationTestHelper()
    .selectBranch('Tallinn - Mitte-Ülemiste Keskus', '1')
    .addWorkingHours('Esmaspäev', 'Laupäev')
    .addWorkingHoursRecord()
    .removeWorkingHoursRecord()
    .getPromise();
});

test ('Dealer can change working hours for holidays', async () => {
  await new ReservationTestHelper()
    .selectBranch('Tallinn - Mitte-Ülemiste Keskus', '1')
    .addHolidayDay()
    .addWorkingHoursForHoliday()
    .addWorkingHoursForHolidayRecord()
    .removeWorkingHoursForHolidayRecord()
    .getPromise()
});

test ('Dealer can change store information', async () => {
  await new ReservationTestHelper()
    .selectBranch('Gagarin', '666')

.addGoogleMapsToStore('https://www.google.com/maps/d/embed?mid=1BRRUC_qMfg3Ja
3Y1zMzBF9dzNYc')
    .addCityToStore('Viimsi', 'string:Viimsi')
    .addStoreAddress('Lauteri tänav 7, A korpus', 'улица Лаутери 7, корпус
A', '7 Lauteri Street, corpus A')
    .addStoreAddressExtraInformation('Ülemiste hotelli kõrval', 'рядом с
отелем Юлемисте', 'near Hotel "Ülemiste"')
    .addStoreExtraInformation('Garantii esindus', 'Гарантийное
представительство', 'Guarantee store')
    .getPromise()
});
```

Allolev koodi lõik näitab kuidas näevad välja meetodid *Helper* klassis:

```
import { Selector, t } from 'testcafe';
import '../conf/process-env.js';

class ReservationTestHelper {
  private user: TestControllerPromise;

  constructor() {
    this.user = t.setTestSpeed(1).maximizeWindow();
  }

  public static async new() {
    return new ReservationTestHelper();
  }

  public selectBranch(branch: string, branchId: string):
ReservationTestHelper {
    const branchSelect = Selector('#selectBranch');
    const branchOption = branchSelect.find('option');

    this.user = this.user
      .click(branchSelect)
      .click(branchOption.withText(branch))
      .expect(branchSelect.value).eql(branchId)
    return this;
  }

  public addWorkingHours(startDate: string, endDate): ReservationTestHelper {
    const newStartDaySelect = Selector('#selectNewStartDay');
    const newStartDayOption = newStartDaySelect.find('option');
    const newEndDaySelect = Selector('#selectNewEndDay');
    const newEndDayOption = newEndDaySelect.find('option');

    this.user = this.user
      .click(Selector('a').withText('Esindusaja haldus'))
      .click(newStartDaySelect)
      .click(newStartDayOption.withText(startDate))
      .click(newEndDaySelect)
      .click(newEndDayOption.withText(endDate))
    return this;
  }
```

Lisa 2 – *Prometheus PromQL*

```
#Reservation rakenduse üldine vigade tase;
errors_total[1m] - Integratsioonis grok_exporter-iga

#Reservation rakenduse HTTP 500, 404, 403, 400 vigade agregatsioon iga 1
minuti jooksul;
sum(rate(http_responses_total{code="500"}[1m])) /
sum(rate(http_responses_total[1m]))
sum(rate(http_responses_total{code="404"}[1m])) /
sum(rate(http_responses_total[1m]))
sum(rate(http_responses_total{code="403"}[1m])) /
sum(rate(http_responses_total[1m]))
sum(rate(http_responses_total{code="400"}[1m])) /
sum(rate(http_responses_total[1m]))

#Reservation rakenduse HTTP 500, 404, 403, 400 vigade agregatsioon nädala
lõikes;
sum(http_responses_total{code="500"} offset 1w)
sum(http_responses_total{code="404"} offset 1w)
sum(http_responses_total{code="403"} offset 1w)
sum(http_responses_total{code="400"} offset 1w)

#teenuse put /reservation/{reservation_id}/cancel „Cancel reservation“ –
teenuse vastuse kiirus ja väljakutsumiste arv;
Counter:
http_requests_total{container="reservation-spring", handler= ~".*/cancel"
}[1m]
Timer:
rate(http_response_time_sum{container="reservation-
spring",handler=~".*/cancel", status="200"}[1m])
/
rate(http_response_time_count{container="reservation-
spring",handler=~".*/cancel", status="200"}[1m])

#teenuse post /reservation/branches/{branch_id} „Create reservation to
branch“, – teenuse vastuse kiirus ja väljakutsumiste arv;
Counter:
http_requests_total{container="reservation-spring", method="POST", handler=
~"/reservation/branches/.*"}[1m]

Timer:
rate(http_response_time_sum{container="reservation-
spring",handler=~"/reservation/branches/.*", status="200"}[1m])
/
rate(http_response_time_count{container="reservation-
spring",handler=~"/reservation/branches/.*", status="200"}[1m])
```

```
#teenuse get /reservation/available/branches/{branch_id} „Get available times
by branch“ – teenuse vastuse kiirus ja väljakutsumiste arv;
Counter:
http_requests_total{container="reservation-spring", method="GET", handler=
~"/reservation/available/branches/.*"}[1m]

Timer:
rate(http_response_time_sum{container="reservation-
spring",handler=~"/reservation/available/branches/.*", status="200"}[1m])
/
rate(http_response_time_count{container="reservation-
spring",handler=~"/reservation/available/branches/.*", status="200"}[1m])

#teenuse get /branches „Get all branches with type STORE and TECH_STORE“ –
teenuse vastuse kiirus ja väljakutsumiste arv;
Counter:
http_requests_total{container="reservation-spring", method="GET", handler=
"/branches"}[1m]

Timer:
rate(http_response_time_sum{container="reservation-
spring",handler="/branches", status="200"}[1m])
/
rate(http_response_time_count{container="reservation-
spring",handler="/branches", status="200"}[1m])

#teenuse get /availbale-slots-by-branches „Find available slots by branches
with type STORE or TECH_STORE“ – teenuse vastuse kiirus ja väljakutsumiste
arv;
Counter:
http_requests_total{job="reservation-spring", method="GET", handler=
"/available-slots-by-branches"}[1m]

Timer:
rate(http_response_time_sum{container="reservation-
spring",handler="/available-slots-by-branches", status="200"}[1m])
/
rate(http_response_time_count{container="reservation-
spring",handler="/available-slots-by-branches", status="200"}[1m])

#99. protsentiil;
histogram_quantile(0.99, sum by (container, le)
(rate(http_request_duration_seconds_bucket[1m])))

histogram_quantile(0.99, sum(rate(http_request_duration_seconds_bucket[1m]))
by (le))
```

Lisaks lähtudes neljast kuldsest signaalist, kus iga monitooringu raames on vaja kindlasti jälgida vigade taset, võrguliiklust, latentsust ja küllastumist (ingl k Saturation) autor lisas ka neid meetrikad Reservation rakenduse monitooringusse.

```
#Võrguliiklus
sum(rate(http_requests_total[1m]))
sum(increase(http_request_total[1m]))


#Latentsus
sum(rate(http_request_duration_seconds_bucket{le="0.4"}[1m])) /
sum(rate(http_request_duration_seconds_count[1m]))

#Küllastumine (ingl k Saturation)
avg_over_time(cpu[1m])
avg_over_time (node_disk_io_time_seconds_total[1m])
```

Lisa 3 – *Zabbix* alarmi konfigureerimine

Zabbix platvormil on olemas kõik live süsteemide hostid juba olemas. Seega konfiguratsioon kriitilise alerdi jälgimiseks näeb välja järgmiselt:

Reservation rakenduse hosti all luuakse uus „Item“, kus „Key“ on kujul

```
custom.ss.plumbr.py[<query_type>,<time_range>,<application_name>]
```

query_type – päritava väärtuse tüüp. Võib olla „app_error_rate“ (kasutatakse frontend rakenduste vea tõenäosuse jaoks) või „api_error_rate“ (kasutatakse teenuste vea tõenäosuse jaoks)
time_range – aja periood mille tagant päritakse error rate väärust. Võib olla 5, 10 minutit või 1 tund ja nii edasi.
application_name – *frontend* või *backend* rakenduse nimi, kirjutatud täpselt niimoodi nagu see on defineeritud *Plumbris*
Valmis kujul „Key“ näeb välja niimoodi:

```
custom.ss.plumbr.py[api_error_rate,10m,reservation]
```

Tagastatav väärtus on Numeric (float)

„Item“ hakkab ainult pärima Reservation error rate andmeid iga 10 minuti tagant. Selleks, et nende päritud andmete alusel hakkaks midagi toimuma on vaja lisada reegli „Trigger“ plokki. „Expression“, mis läheb trigeri reeglisse näeb välja niimoodi:

```
{HOST:item_key.min(#3)}>3
```

HOST – rakenduse nimetus (*CI*) konfiguratsiooni juhtimise andmebaasis (Configuration management database, *CMDB)*
item_key – see on „Key“, mis oli defineeritud „Item“ plokis ehk custom.ss.plumbr.py[api_error_rate,10m,reservation]
min(#3) – miinimum 3 korda

>3 – rohkem kui 3

Lõplik „Expression“ näeb välja niimoodi:

```
{HOST:custom.ss.plumbr.py[api_error_rate,10m,reservation].min(#3)}>3
```

Lisa 4 – Automaatne tagasipööre eelmisele versioonile (1 väärtus)

```
#!/bin/bash

set -e
#autentification and retrive data

for (( ; ; ))
do
   echo "Start to request Plumbr"

DEPLOYMENT_PROJECT="$1"
ENVIRONMENT_ID="$2"
PLUMBR_APPLICATION="$3"


start=$(date +"%Y-%m-%dT%H:%M - 10 minutes")
echo $start
end=$(date +"%Y-%m-%dT%H:%M")
echo $end


errorRate=&(curl -v -u test:psw
'http://test.ee:1080/api/errors/recent?startDate={$start}&endDate={$end}&filt
er=applicationName%3D{$PLUMBR_APPLICATION}')
echo $errorRate


#if error rate is more th%
# ask for the name of current deploy version
# ask for the list of last releases
# do deploy for the previous deployed version
# resources: bamboo api https://docs.atlassian.com/atlassian-
bamboo/REST/6.1.1/
# https://developer.atlassian.com/server/bamboo/bamboo-rest-resources/

if [ "$errorrate" -gt 2 ];
  then
    #if we need to do rollback in live environment, we should find out
previous release
    previousPrimaryRelease="$(curl
"https://test.net/rest/api/latest/deploy/project/${DEPLOYMENT_PROJECT}/versio
ns" -u test_login:${bamboo.ad.user.password} -H "Accepts: application/json" |
jq '.versions[1] | {id: .id}')"
```

```
    currentPrimaryRelease="$(curl
"https://test.net/rest/api/latest/deploy/project/${DEPLOYMENT_PROJECT}/versio
ns" -u test_login:${bamboo.ad.user.password} -H "Accepts: application/json" |
jq '.versions[0] | {id: .id}')"

  if [ "$ENVIRONMENT_ID" == "canary" ]
    then
    #if it is canary release
    revertToPreviousPrimaryVersion="$(curl -X POST
"https://test.net/rest/api/latest/queue/deployment/?environmentId=${ENVIRONME
NT_ID}&versionId=${currentPrimaryRelease}" -u
test_login:${bamboo.ad.user.password} -H "Accepts: application/json" | jq
'.deployments[0] | .deploymentResultId ')"
    checkVersionReversion="$(curl
"https://test.net/rest/api/latest/deploy/result/${revertToPreviousPrimaryVers
ion}" -u test_login:${bamboo.ad.user.password}-H "Accepts: application/json"
| jq '.')"

  elif [ "$ENVIRONMENT_ID" == "primary" ]
    then
    #if it is live release
    revertLiveToPreviousLiveVersion="$(curl -X POST
"https://test.net/rest/api/latest/queue/deployment/?environmentId=${ENVIRONME
NT_ID}&versionId=${previousPrimaryRelease}" -u
test_login:${bamboo.ad.user.password} -H "Accepts: application/json" | jq
'.deployments[0] | .deploymentResultId ')"
    checkLiveVersionReversion="$(curl
"https://test.net/rest/api/latest/deploy/result/${revertLiveToPreviousLiveVer
sion}" -u test_login:${bamboo.ad.user.password}-H "Accepts: application/json"
| jq '.')"
  fi

  rollbackMessage="WARNING: version ${currentPrimaryRelease} was changed to
${previousPrimaryRelease} due to high error rate"
  echo "$rollbackMessage"
fi

done
sleep 10m

done
echo "Done."
```

Lisa 5 – Automaatne tagasipööre eelmisele versioonile (3 väärtust)

```
#!/bin/bash

set -e
#autentification and retrive data

for (( ; ; ))
do
   echo "Start to request Plumbr"

start=$(date +"%Y-%m-%dT%H:%M - 10 minutes")
echo $start
end=$(date +"%Y-%m-%dT%H:%M")
echo $end

DEPLOYMENT_PROJECT="$1"
ENVIRONMENT_ID="$2"
PLUMBR_APPLICATION="$3"

for a in {1..3}; do
lastErrorRates=()

errorRate=&(curl -v -u test:psw
'http://test.ee:1080/api/errors/recent?startDate={$start}&endDate={$end}&filt
er=applicationName%3D{$PLUMBR_APPLICATION}')
echo $errorRate

lastErrorRates+=("$errorRate")

# average =  sum of numbers / array size

IFS='+' sumOfLastErrorRates=$(echo
"scale=1;(${lastErrorRates[*]})/${#lastErrorRates[@]}"|bc)
echo $sumOfLastErrorRates

#if error rate is more than 2% for 3 times
# ask for the name of current deploy version
# ask for the list of last releases
# do deploy for the previous deployed version
# resources: bamboo api https://docs.atlassian.com/atlassian-
bamboo/REST/6.1.1/
# https://developer.atlassian.com/server/bamboo/bamboo-rest-resources/
```

```
if [ "${#lastErrorRates[@]}" == 3 ] && [ "$sumOfLastErrorRates" -gt 2 ];
  then
    #if we need to do rollback in live environment, we should find out
previous release
    previousPrimaryRelease="$(curl
"https://test.net/rest/api/latest/deploy/project/${DEPLOYMENT_PROJECT}/versio
ns" -u test_login:${bamboo.ad.user.password} -H "Accepts: application/json" |
jq '.versions[1] | {id: .id}')"
    currentPrimaryRelease="$(curl
"https://test.net/rest/api/latest/deploy/project/${DEPLOYMENT_PROJECT}/versio
ns" -u test_login:${bamboo.ad.user.password} -H "Accepts: application/json" |
jq '.versions[0] | {id: .id}')"

  if [ "$ENVIRONMENT_ID" == "canary" ]
    then
    #if it is canary release
    revertToPreviousPrimaryVersion="$(curl -X POST
"https://test.net/rest/api/latest/queue/deployment/?environmentId=${ENVIRONME
NT_ID}&versionId=${currentPrimaryRelease}" -u
test_login:${bamboo.ad.user.password} -H "Accepts: application/json" | jq
'.deployments[0] | .deploymentResultId ')"
    checkVersionReversion="$(curl
"https://test.net/rest/api/latest/deploy/result/${revertToPreviousPrimaryVers
ion}" -u test_login:${bamboo.ad.user.password}-H "Accepts: application/json"
| jq '.')"

  elif [ "$ENVIRONMENT_ID" == "primary" ]
    then
    #if it is live release
    revertLiveToPreviousLiveVersion="$(curl -X POST
"https://test.net/rest/api/latest/queue/deployment/?environmentId=${ENVIRONME
NT_ID}&versionId=${previousPrimaryRelease}" -u
test_login:${bamboo.ad.user.password} -H "Accepts: application/json" | jq
'.deployments[0] | .deploymentResultId ')"
    checkLiveVersionReversion="$(curl
"https://test.net/rest/api/latest/deploy/result/${revertLiveToPreviousLiveVer
sion}" -u test_login:${bamboo.ad.user.password}-H "Accepts: application/json"
| jq '.')"
  fi

  rollbackMessage="WARNING: version ${currentPrimaryRelease} was changed to
${previousPrimaryRelease} due to high error rate"
  echo "$rollbackMessage"
fi

done
sleep 10m

done
echo "Done."
```

Lisa
rektori 07.04.2020 käskkirjale nr 1-8/17

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina Jelena Sarap

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
AUTOMATISEERITUD KVALITEEDIKONTROLLI JA TAGASIPÖÖRDE VÕIMALUSEGA PIDEVJUURUTUSE PROTSESSI LOOMINE TELIA EESTI AS NÄITEL,

mille juhendaja on Maili Markvardt,

1.1 reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2 üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.

3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

---

10 mai 2021 (kuupäev)

[1] *Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.*