TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Andree Kuriks 193465IADB

DevOps mõõdikute kogumine Kubernetese platvormil

Bakalaureusetöö

Juhendaja: Rein Remmel
Rakenduskõrgharidus

Tallinn 2023

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: Andree Kuriks

15.05.2023

Annotatsioon

Lõputöö eesmärgiks on luua rakendus, mis analüüsib Kubernetese platvormil paikneva süsteemi arendusprotsessi, mille tulemusena saadakse väärtused vastavalt DevOps mõõdikutele. Devops mõõdikud kirjeldavad arendava organisatsiooni või meeskonna võimekust edukalt tarnida, mis omakorda loob eeldused tootlikkuseks. Loodava rakenduse olemasolul kaob vajadus ettevõtetel ja tiimidel üles seada manuaalseid raporteerimise protsesse.

Keerukas on hinnata arendusprotsessi ja DevOps põhimõtete järgimist organisatsioonis. Manuaalne ülevaatus on ebatäpne ja aeganõudev. Arendusprotsessi võib kuuluda mitmeid keskkondi, platvorme ja komponente, millest on vajalik andmeid koguda täpse hindamise jaoks. DevOps põhimõtete järgimine ja hea arendusprotsessi olemasolu võib olla määrav, et olla teiste ettevõtetega konkurentsivõimeline.

Lõputöö lahendus on pilootkatse vormis ja proovitakse välja selgitada, kas eesmärki on võimalik täita ja kas edasiseid investeeringuid on mõistlik teha. Luuakse prototüüp, millega valideeritakse jätkuarenduste mõistlikkus.

Loodav rakendus paikneb Kubernetese klastris ja kogub infot seoses paigalduse sündmustega. Paigalduse sündmuse toimumisel määratakse paigalduse tüüp ja kogutakse vajalikud andmed DevOps mõõdikute arvutamiseks. Lõputöö skoop on piiratud ainult Kubernetese platvormilt saadavate andmetega ning koodihoidla ja tööprotsessi piletite haldamise süsteemi integratsioon jääb lahendusest välja.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 39 leheküljel, 6 peatükki, 9 joonist, 1 tabelit.

Abstract

DevOps Metrics Gathering on Kubernetes Platform

The aim of the thesis is to create an application that analyzes the development process of a Kubernetes-based system, as a result of which values are obtained and aggregated to DevOps metrics. DevOps metrics describe the ability of the developing team or organization to deliver successfully and thus creates the prerequisites for productivity. As a result, there is a tool that helps companies find bottlenecks in their development processes and thereby increase performance. The application helps managers keep an eye on work processes, which in turn contributes to the delegation of decision-making authority to teams, which is one of the prerequisites of DevOps practices. With the creation of the application, the need for companies and teams to set up manual reporting processes disappear.

It is difficult to assess the development process and compliance with DevOps principles within an organization. Manual inspection is imprecise and time-consuming. The development process may include several environments, platforms and components from which it is necessary to collect data in order to fulfill the purpose. Following DevOps principles and having a good development process can be decisive to be competitive with other companies.

The solution of the thesis is in the form of a proof of concept and an attempt is made to find out whether the goal can be met and whether it is reasonable to make further investments. A prototype is created to validate the reasonableness of further developments. The integration of code repository and ticket management system is left out of the prototype. The created application will be located in a Kubernetes cluster and will listen to events related to deployments. When a new software release occurs in the cluster, the deployment type is determined and the necessary data is collected to calculate the DevOps metrics.

The thesis is in Estonian and contains 39 pages of text, 6 chapters, 9 figures, 1 table.

Lühendite ja mõistete sõnastik

ETCD	Klastri andmekogum, koosneb võti-väärtus paaridest
Event	Kubernetese objekt, mis tähistab mõne objekti olekumuutust või sündmust klastris
Kaun	*Pod,* Kubernetese klastris paiknev objekt, mis sisaldab rakenduse konteinereid ja seob need omavahel ühtseks komponendiks
Koodifikseering	*Commit,* koodi fikseerimine versioonihalduse keskkonda
OLAP	*On-line analytical processing,* andmetöötluse tehnoloogia, mis toetab keerulisi analüütilisi päringuid organisatsiooniliste otsuste tegemiseks ilma transaktsioonilisi süsteeme häirimata
OLTP	*On-line transaction processing,* andmetöötluse tehnoloogia, mis toetab edukalt transaktsioonipõhiseid rakendusi ja samaaegseid päringuid
Deployment	Kubernetese objekt, millega deklareeritakse objektide kooslus ehk rakendus. Loob replikeerimiskogumi (ingl *Replicaset*), mis omakorda haldab loodud kaunasid
Paigaldus	Tehniline sündmus, lisatakse või muudetakse komponenti
Pilootkatse	*Proof of concept,* katse tulemusena tõestatakse idee ja selle realiseerimise võimalikkus ja mõistlikkus
Reliis	Äriline sündmus, kliendile võimaldatakse kasutada uut funktsionaalsust
Replikeerimiskogum	*Replicaset,* Kubernetese objekt, mis garanteerib kindla fikseeritud arvu kaunade kättesaadavuse ning hoiab püsivat ja stabiilset olekut ja kogumit tema alla käivatest kaunadest
Sidusarendus	*Continuous integration, continuous delivery (CICD)*, riskide vähendamisele, agiilsusele, töövoo sujuvusele ja automatiseerimisele suunatud arendustöö viis.
Sõlm	*Node*, Kubernetese klastris paiknev objekt. Hulk eraldatud ressursse, mis võimaldab Kubernetese objektide majutamist
Toodang	*Production*, keskkond, kus tarkvara hilisemad versioonid klientidele kättesaadavaks tehakse. Testimised ja vigade parandused on eelnevalt sooritatud.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

1.1 Taust ja probleem

Enamus IT ettevõtteid kasutavad täna DevOpsi kui kultuuri, põhimõtete ja töövahendite kooslust, millega tõsta tulemuslikkust ja pakkuda väärtust kliendile kiiresti, järjepidevalt ja turvaliselt. Kuid DevOps põhimõtete järgimine ja arendusprotsessi hindamine on tihti keerukas – manuaalne ülevaatamine on ajakulukas ning tihtipeale ei teata, kuidas keerukat arendusprotsessi ja selle mitmeid keskkondi, platvorme ja komponente kokku tuua ja detailselt hinnata. See on aga vajalik, et olla agiilne ja konkureeriv teiste ettevõtetega. Suurimad ettevõtted, nagu näiteks Netflix, teevad kümneid tuhandeid koodi paigaldusi (ingl *deployment*) päevas [1]. See on võimalik tänu DevOps tavade järgimisele.

Lahendus baseerub DevOps mõõdikute kogumisel läbi tarkvaraarenduse protsessi analüüsimise, mille abil saab hinnata organisatsiooni või meeskonna võimekust edukalt tarnida mis omakorda loob eeldused tootlikkuseks. Need mõõdikud tulenevad DORA (Google Cloud's DevOps Research and Assessment) aruandest nimega „2022 Accelerate State of Devops Report" [2].

Arendusprotsesside efektiivne hindamine on keerukas töö, sest sõltuvusi on mitmeid – esineb erinevaid arendusmetoodikaid ja tavasid organisatsioonides ning erinevaid arendusplatvorme ja keskkondi. Sellest tulenevalt on lõputöö peamiselt pilootkatse vormis (ingl *proof of concept*, POC), mille eesmärk on kinnitada idee realiseerimise võimalikkus ja kasulikkus.

1.2 Eesmärk

Lõputöö eesmärk on luua rakendus, mis analüüsib Kubernetese platvormil paikneva süsteemi arendusprotsessi, mille tulemusena saadakse väärtused vastavalt DevOps mõõdikutele. DevOps mõõdikud kirjeldavad arendava organisatsiooni või meeskonna võimekust edukalt tarnida ja seeläbi loob eeldused tootlikkuseks.

Eesmärk on luua prototüüp, millega valideeritakse jätkuarenduste mõistlikkus. Seda põhjusel, et rakenduse valmimisel on olemas vahend, mis aitab ettevõtetel ja meeskondadel tõsta tulemuslikkust, leides kitsaskohti arendusprotsessides. Lisaks, aitab rakendus ettevõtte juhtidel silma peal hoida tööprotsessidel, mis omakorda aitab kaasa otsustusõiguse alla poole meeskondadesse delegeerimisele, mis on üks DevOps tavade eeldus. Seda kõike ilma, et ettevõtted ja tiimid peaksid üles seadma manuaalseid raporteerimise protsesse.

2 Taust ja teoreetilised alused

2.1 Metoodika

Aluseks on kvalitatiivne analüüs, kus lähtutakse enamasti viimase kümnendi jooksul DevOps valdkonnas loodud teadmistele ja kogemustele raamatute ja artiklite näol. Selgitatakse lahti DevOps ja selle mõõdikute tausta ja mõõtmise vajadust toetudes kirjandusele. Analüüsitakse erinevaid keskkondi ja platvorme, mida kasutatakse IT-süsteemide loomisel (näiteks Kubernetes, Docker) ja uuritakse võimalikke viise info kogumiseks. Uuritakse teisi sarnanevaid lahendusi ning luuakse loogika, kuidas ja millist infot koguda, et saaks täisväärtusliku pildi arendusprotsessist vastavalt DevOps mõõdikutele.

2.2 DevOps

Kogu IT ja tarkvaraarenduse keskmes on kiirus ja tootlikkus. DevOps on esile kerkinud kui kultuuri, põhimõtete ja töövahendite kooslus, millega luua innovatiivseid tooteid ja tarnida teenuseid turule kiiremini ning seeläbi suurendada tootlikkust ja kasumlikkust. DevOps ühendab omavahel tarkvaraarenduse ja operatsioonide maailmad, saavutades selle läbi automatiseeritud arendusprotsesside ning infrastruktuuri monitooringuga. [3]

Minevikus käsitleti arendust ja operatsioone täiesti eraldiseisvana – omavahelist suhtlust ja tagasisidet teiselt poolelt oli minimaalselt. Tänapäevane tarkvaraarendus peab käsitlema operatsioone kui signaalide allikana arendusmeeskondadele. [4]

DevOps hõlmab enda alla organisatsioonilise muutuse, kus erineva otstarbega eraldiseisvate meeskondade asemel esinevad ristfunktsionaalsed meeskonnad. Selline lähenemine aitab kaasa kiiremale ja järjepidevamale arendusele, sest see kiirendab probleemide lahendamist eri meeskondade vahel ja vähendab puudulikust või ebatäpsest suhtlusest tulenevaid probleeme [3]. Lisaks, on organisatsiooni investeeringute osakaal DevOps-i kõrgelt korrelatsioonis tarkvara tarne tootlikkusega [5].

2.3 DevOps mõõdikud

DevOps mõõdikud on kogum andmeid, mis mõõdavad arendava organisatsiooni võimekust edukalt tarnida ning aitavad identifitseerida kitsaskohti arendusprotsessides. Mõõdikud tulenevad Google uurimisrühma DORA (DevOps Research and Assessment LLC) aastatepikkusest uuringust, mille tulemusena on tuvastatud neli mõõdikut, mis kõige täpsemini ja ülevaatlikumalt kirjeldavad ja hindavad organisatsiooni või tiimi võimekust edukalt tarnida, mis omakorda loob eeldused tootlikuseks. [5]

2.3.1 Paigalduste sagedus

Paigalduste sagedus (ingl *deployment frequency*) kirjeldab, kui sagedaselt uut koodi toodangusse paigaldatakse. Mõõdik tuleneb Lean paradigma partii suuruse mõõdikust, mis oli üks edu alustest Toyota tootmissüsteemis. Partii suuruse vähendamine lühendab tootmistsükli aegasid, tootmisvoo muutlikkust, teadmatusest tulenevaid riske ning parendab tagasisidet, tootlikkust, töötajate motivatsiooni ja vähendab kulusid. Tarkvara kontekstis on partii suurust keerukas mõõta, sest pole otsest inventuuri. Sellest tulenevalt, DORA aruande autorid tõid IT analoogiks paigalduste sageduse. [5]

Efektiivseimad meeskonnad paigaldavad tarkvara mitu korda päevas, erinevalt ebaefektiivsetest meeskondadest, kes paigaldavad korra ühe kuni kuue kuu jooksul, ehk 208 korda sagedamini kui ebaefektiivseimad meeskonnad. Näiteks Netflix, Amazon ja Google paigaldavad tuhandeid kordi päevas. [1]

2.3.2 Muudatuste jõustumise aeg

Muudatuste jõustumise aeg (ingl *lead time for changes*) kirjeldab, kui kaua läheb aega, et koodimuudatused ja arendused edukalt toodangusse jõuaksid. Koos paigalduste sageduse mõõdikuga käib muudatuste jõustumise aja mõõdik meeskonna ja selle protsessi kiiruse kirjeldamise alla.

Efektiivseimate meeskondade puhul jääb selle mõõdiku keskmiseks väärtuseks vähem kui 1 päev. Ebaefektiivseimate meeskondade puhul on selle väärtuseks 1 kuu kuni 6 kuud. Tegu on kuni 106 kordse vahega. [1]

2.3.3 Teenuse taastamise aeg

Teenuse taastamise aeg (ingl *time to restore service*) kirjeldab, kui kaua läheb aega, et teenus vigasest seisundist stabiilsesse seisundisse taastada. Modernsete kompleksete süsteemide puhul on vead paratamatud, seega küsimus pole, kas vigu tekib, vaid kui kiiresti süsteem taastatakse. Siinkohal ei mängi rolli, kas vigane seisund tekkis eelneva paigalduse pärast või seoses millegi muuga. Mõõdik kirjeldab meekonna võimet tagada arendatava teenuse stabiilsus ja töökindlus.

Efektiivseimate meeskondade puhul on teenuse taastamise ajaks keskmiselt vähem kui 1 tund ning ebaefektiivseimate meeskondade puhul keskmiselt 1 nädal kuni 1 kuu. See teeb kuni 2604 kordse vahe. [1]

2.3.4 Muudatuste ebaõnnestumise määr

Muudatuste ebaõnnestumise määr (ingl *change failure rate*) kirjeldab, kui palju koodimuudatustest, mis on jõudnud toodangusse, on vigased ja nõuavad parandust. Koos teenuse taastamise aja mõõdikuga kirjeldab muudatuste ebaõnnestumise määr meeskonna ja selle protsessi stabiilsust ja töökindlust.

Efektiivseimate meeskondade puhul on muudatuste ebaõnnestumise määr keskmiselt 0% ja 15% vahel. Ebaefektiivseimate meeskondade puhul on selleks arvuks 46% kuni 60%, ehk tegu on kuni 7 kordse vahega. [1]

2.4 Kubernetes

Lõputöös loodud rakendus paikneb Kubernetese klastris ja reageerib klastris toimuvate muutuste peale. Kubernetes on avatud lähtekoodiga orkestreerimise vahend konteineriseeritud rakenduste jaoks. Kubernetes võimaldab edukalt ehitada ja paigaldada süsteeme, mis on skaleeritud, hajutatud ja töökindlad. [6]

Tooted ja teenused mida pakutakse interneti vahendusel muutuvad aina laialdasemaks ja tähtsamaks inimeste igapäevaelus ning sellest tulenevalt peavad need olema aina töökindlamad. Käideldavus peab olema tagatud juhul, kui tehakse regulaarseid hooldustoiminguid või siis, kui osa süsteemist jookseb kokku. Lisaks, peavad need süsteemid olema ka skaleeritavad, et vajalik klientide maht oleks rahuldatud ilma, et peaks radikaalseid ümbermuudatusi tegema süsteemi disainis. Sama ka vastupidises

olukorras, kus süsteemi majutatuvuse maht kahaneb sekunditega, et süsteem ja selle komponendid oleksid võimalikult otstarbekalt ja kokkuhoidlikult kasutatud [6]. Kubernetese kasutuselevõtt võib suuresti vähendada organisatsiooni kulusid ja kiirendada turule jõudmise aega [7].

2.4.1 Kubernetese turuosa

Kubernetes on alates 2014. aastast, millal ta avalikkuse ette toodi, tõusnud üheks populaarseimaks ja suurimaks avatud lähtekoodiga projektiks maailmas. Sellest on saanud põhiline API pilvepõhiste rakenduste ehitamiseks ja haldamiseks [6]. 2022 aastal hinnati Kubernetese turu suuruseks 1.8 miljardit dollarit (USD) ning prognoositavalt tõuseb see 2030. aastaks 7.8 miljardini. See tuleneb avaliku lähtekoodiga konteineri platvormide pakkujate arvukuse tõusust ja investeeringutest nendesse, mikroteenuste populaarsuse tõusust, digitaalsete transformatsioonide sagedusest ettevõtetes ja vajadusest järgida regulatiivseid nõudeid. [7]

2.4.2 Kasutatavad komponendid

Kubernetese klastris paiknevad sõlmed (ingl *node*). Sõlmede töö on jooksutada enda sees paiknevaid konteineriseeritud rakendusi, mis omakorda paiknevad eraldi sõlme enda poolt majutatud kaunades (ingl *pod*). [8]

Kubernetese *Deployment* objektid haldavad objektis deklareeritud konteineriseeritud rakendusi. *Deployment* loob replikeerimiskogumi, mis haldab kauna ja selle arvukust, mis on *Deployment* objektis kirjeldatud. Täpsemini, on replikeerimiskogumi ülesanne hoida püsivat ja stabiilset olekut *Deployment*-is deklareeritud kaunal. [9]

Event objektid annavad informatsiooni kõiksuguste klastris toimuvate sündmuste ja muudatuste kohta. Nendeks sündmusteks võivad olla näiteks uue kauna teke, Docker tõmmise alla tõmbamise alustamine, tõmmise eduka alla tõmbamise lõpetamine.

Lõputöös loodud rakendus paikneb Kubernetese klastris eraldi kaunana. Rakendus tegeleb Kubernetese *Deployment-ide*, kaunade ja *Event* objektide jälgimisega ning kogub mõõdikuid huvipakkuvate sündmuste esinedes.

2.5 Docker ja konteinerid

Docker on avatud lähtekoodiga platvorm, mis pakub võimaluse ehitada, pakendada ja väljastada rakendusi konteineritena. Konteineriseerimine on üks virutaliseerimise võimalusi. Erinevalt virtuaalmasinatest, mis virtualiseerivad võõrustaja (ingl *host*) riistvara, virtualiseerivad konteinerid võõrustaja operatsioonisüsteemi. Samal ajal on konteinerid isoleeritud teistest võõrustaja masinal paiknevatest protsessidest. [10]

Konteinerid on palju kompaktsemad kui virtuaalmasinad. Nende jaoks ei ole vaja eraldi hüperviisorit (tarkvara, mis haldab virtuaalmasinaid ning loob ja käivitab neid) võõrustaja masinasse ja tervet operatsioonisüsteemi ja selle sõltuvusi, et jooksutada ühte rakendust. Konteinerid sisaldavad endas kõike, et käivitada rakendus minimaalsete ressurssidega. Neid on kerge ja kiire paigaldada ning sellel põhjusel on nad kasulikud DevOps tavade rakendamisel ja sidusarenduse elluviimisel. [10]

2.5.1 Kasutatavad komponendid

Kubernetese klastrisse saab paigaldada kaunasid (ingl *pod*), mis sisaldavad konteinereid. Konteiner on ehitatud Docker tõmmise (ingl *image*) järgi, mis on nagu juhis, mida ei saa muuta ning mis kirjeldab, kuidas konteinerit ehitada ja mida see sisaldama peab.

Tõmmisel on alati juures ka versioon (ingl *tag*), mis märgistab tõmmise ja aitab neid eristada. Loodud rakenduses kasutatakse tõmmise versiooni muutuseid, et tuvastada ja eristada sündmuseid (uuendus, kiirparandus või tagasipööramine). Tõmmise versioon ei ole alati täpne viis identifitseerimiseks. Versioon on lihtsalt inimloetav alias, mis ei võta kuidagi arvesse tõmmise reaalset sisu ja selle muutust. Näiteks tihti kasutatav "*latest*" versioon on pidevalt muutuv. Et täpsemalt tõmmiseid eristada, tuvastada ja leida, on igal tõmmisel olemas ka SHA256 krüpteeringuga räsi (ingl *image digest*). Tõmmise räsi on alati sama juhul, kui sisendfailid ja sõltuvused on muutmata. Ehk tõmmise räsi järgi saab alati määrata reaalse muudatuse toimumise. [11]

Docker-iga on kaasas ka käsurea liides, millega saab lokaalseid päringuid teha. Lõputöös loodud rakenduses on käsurea liides vajalik kahe eesmärgi täitmise jaoks:

1. Pärida Docker tõmmise metaandmetest välja tõmmise loomise kuupäev

2. Pärida Docker tõmmise metaandmetest välja tõmmise räsi

2.6 SemVer versioniseerimine

SemVer on populaarne semantilise versioniseerimise standard, mida kasutatakse sagedaselt projektides, et identifitseerida ja klassifitseerida versiooni väljalaskeid. Versioniseerimise muster on kujul X.Y.Z ning see koosneb kolmest osast – *major* versioon, *minor* versioon ja *patch* versioon. Major versiooni suurendatakse, kui luuakse mitte-tagasiühilduvaid muudatusi, *minor* versiooni suurendatakse, kui luuakse tagasiühilduvaid muudatusi ja *patch* versiooni suurendatakse, kui luuakse tagasiühilduvaid veaparandusi. [12]

Lõputöös loodud rakendus töötab täpsemini, kui loodud Docker tõmmistes on kasutatud Semver versioniseerimist. Semver versioniseerimise järgi saab turvaliselt määrata tekkivate paigalduse sündmuste tüüpe (uuendus, kiirparandus, tagasipööramine) ja teha seejärel vastavaid arvutusi ja mõõdikute liigitamist.

3 Analüüs

Andmete kogumine DevOps mõõdikute arvutamiseks võib tarkvaraarenduse protsessi puhul keerukas olla, sest tänapäeval on kasutusel väga palju erinevaid tehnoloogiaid, pilvi ja süsteeme protsessi läbimiseks. Lisaks on mitmeid eri otstarbega meeskondi, kellel on erinevad keskkonnad ja protsessid kasutusel. Andmed peavad äriliste otsuste tegemiseks samal ajal olema ka terviklikud, laiaulatuslikud, ja korrektsed. Tuleb olla tähelepanelik, et andmed erinevatest platvormidest oleks korreleeritud ehk poleks topelt lugemist (ühesuguse mõjuga andmed mitmes kasutusel olevas platvormis), andmete kogumine tuleneks kõigist vajaminevatest komponentidest ning, et piisavalt andmeid oleks kogutud. [13]

3.1 Paigalduste sagedus

Paigalduste sagedus annab informatsiooni tarkvaraarenduse protsessi kiiruse kohta. Kui paigalduste sagedus on liiga madal, võib see tähendada näiteks probleeme koodi ülevaatusega, testimisega, tarkvara tarne süsteemiga, või sellega, et korraga võetakse liiga suur tükk tööd ette.

Mõõdik on definitsiooni järgi lihtne – " paigalduste arv toodangusse kindlal ajaperioodil", kuid esineb variatsioone. Mõningad lahendused loevad paigalduse alla ainult edukat paigaldust. Sellise lähenemisega tuleb defineerida, mis tähendab "edukas" paigaldus. Vaatamata sellele, kuna tegu on kiirust määrava mõõdikuga, ei tehta loodud rakenduses mõõdiku arvutamisel vahet edukal ja mitteedukal paigaldusel. Vajalik on teha kindlaks kuidas paigaldust identifitseerida. [14]

3.1.1 Võimalikud lahendused

Raamatus "Agile Processes in Software Engineering and Extreme Programming" toodud lahenduses tuvastatakse Docker tõmmise registrist tõmbamise sündmuseid OpenShift konteineri platvormil, mis fikseerib paigalduse toimimise [14]. Eksisteerib rakendusi, kus paigalduse tuvastamiseks jälgitakse Github-i veebihaake (ingl *webhook*) ja ehitustsükli käivitusi [15].

3.1.2 Loodav lahendus

Koodihoidla veebihaagid ja ehitustsüklite käivitamised on konkreetsed ja täpsed viisid paigalduse sündmuse tuvastamiseks, kuid nõuavad eraldi konfigureerimist rakenduse siseselt kui ka väliselt. Soovitud on leida lahendus, kus kõik vajalik mõõdiku kogumiseks on võimalik leida minimaalse või ideaalis ilma igasuguse eraldi rakenduse konfigureerimiseta. Eesmärk on rakendada loodud programm Kubernetese platvormile ja seal toimuvate sündmuste jälgimisele, seega kogu protsess käib läbi Kubernetese objektide ja nendes sisalduvate Docker tõmmiste, läbi mille saab tuvastada paigalduse sündmuseid ja nende tüüpe.

Paigaldus tekib, kui Kubernetese klastris tuvastatakse *Deployment* objektis sisalduva konteineri tõmmise muutus. Kui tegu on uue *Deployment*-i tekkega klastris, loetakse iga selle objekti konteinerit eraldi paigalduseks. Kui tegu on aga olemasoleva *Deployment* objekti muutumisega, tuvastatakse objekti kõik uued ja muutunud konteinerid (muutumine fikseeritakse, kui kasutatud Docker tõmmise versioon on muutunud) ja iga üks neist on eraldi paigaldus. Paigalduste sageduseks on igat tüüpi paigalduse (uuendus, kiirparandus, tagasipööramine) sagedus kindlas ajaühikus.

3.2 Muudatuste jõustumise aeg

Sarnaselt paigalduste sagedusele, annab muudatuste jõustumise aja mõõdik teavet arendusprotsessi kiiruse kohta. Lühike muudatuste jõustumise aeg näitab, et arendusprotsess on efektiivne, sujuv ja takistusteta. Kui aga vastupidi, siis muudatuste jõustumise aja mõõdik aitab tuvastada kitsaskohti näiteks testimisega ja koodi ülevaatusega.

Vajalik on kindlaks määrata, millal tehti esimene muudatus (näiteks esimene koodifikseering ehk *commit*). Seejärel tuleb kindlaks teha, mis on muudatuse jõustumine (näiteks paigaldus Kubernetese klastrisse Docker tõmmise näol) ja millal see toimus. Mõõdiku saamiseks tuleb arvutada nende kahe sündmuse ajaline vahe.

Mõõdiku defineerimine on olemasolevates taolistes lahendustes üldiselt üheselt mõistetav, erinevusi esineb ainult tulemuse agregeerimises – kas mediaan, keskmine või mingi muu kuju. [14]

3.2.1 Võimalikud lahendused

Docker tõmmise registrist tõmbamise sündmuses on kirjas tõmmise nimi ja versioon. Giti koodifikseeringu räsi loetakse sisse vastavast artefaktist Docker tõmmise nime ja versiooni abil ning Git repositooriumi internetiaadress on kinnitatud Docker tõmmise metaandmetesse. Seejärel on saadud andmete põhjal iga uue koodifikseeringu kohta toodangusse võimalik arvutada muudatuste jõustumise aeg. [14]

Teiste näidete puhul eksisteerib lahendusi, kus eduka paigalduse korral mõõdetakse aeg viimasest koodifikseeringust kuni liitumise päringuni (ingl *merge request*). Eelduseks on, et liitumise päringu korral tekib automaatne paigaldus ja kasutatakse maksimaalselt ühte haru muudatuste elluviimisel, sest jälgitakse kõiki koodifikseeringuid, mis ei paikne pea harus ehk tüves. Seega on ka eelduseks, et arendus käib alati ühe featuuri haru kaupa. [15]

3.2.2 Loodav lahendus

Lõputöö on pilootkatse raames ning vaja on tõestada idee realiseerimise võimalikkus ja mõistlikkus. Git ja muude koodihoidlate integreerimine rakendusse jääb lõputöö skoopist välja, sest täpselt töötava lahenduse loomine on liiga aeganõudev – esineb erinevaid harjumusi ja mustreid arendusstiilides seoses koodihoidlate kasutamisega, nagu näiteks *Git flow* ja *trunk-based*. Võimalused lähtekoodihoidlaga suhtlemiseks on siiski olemas – kasutada saab koodihoidlate avalikke rakendusliideseid (API) ja mitmeid teeke, mis suhtluse koodihoidlaga läbi rakendusliidese lihtsustab. Selline funktsionaalsus on tulevikus võimalik loodavale rakendusele peale ehitada.

Muudatuste jõustumise aja mõõtmiseks kasutatakse koodihoidla koodifikseeringute analüüsimise asemel klastrisse tekkivate Docker tõmmiste loomise aegu. Selline lahendusviis ei mõõda täpselt muudatuse jõustumise aega, vaid uue versiooni ehk valminud koodi komplekti kasutuselevõtu aega toodangu keskkonnas, kuid pilootkatse raames on selline lahendus rahuldatav.

Lahenduses kasutatakse paigalduse sündmuse ja *Event* objekti teket, mis luuakse pärast Docker tõmmise klastrisse alla tõmbamist. *Event* objekti tekke jälgimine on oluline, et pärida tõmmise metaandmetest välja tõmmise loomise aeg, sest alles sellest hetkest alates on võimalik lokaalselt Docker-i käsurea liidesega metaandmeid pärida. Need kaks erinevat juhtumit - tuvastatud paigalduse sündmus ja *Event* objekt, seotakse omavahel

klastrisse tekkinud kauna UID (*unique identifier*) ja Docker tõmmise nime ja versiooni järgi (kauna UID on *Event* objektis kirjas). Pärast sidumist loetakse lokaalse tõmmise metaandmetest välja tõmmise loomise kuupäev. Muudatuste jõustumise ajaks on tõmmise klastrisse alla laadimise pealt käivitatud tööprotsessi ja tõmmise ehitamise ajaline vahe.

Antud lahendus töötab, kui loodud rakendus ja *Deployment* objekti poolt käivitatav kaun eksisteerivad samal sõlmel. Seda põhjusel, et Docker tõmmis laetakse alla kindlale sõlmele, ning ei ole üle klastri kättesaadav. Suuremate klastrite jaoks on vaja lahendust täiendada – näiteks „Muudatuste jõustumise aeg“ mõõdiku arvutamisel pärida vajaminevaid andmeid koodihoidlast.

3.3 Teenuse taastamise aeg

Teenuse taastamise aja mõõdik annab informatsiooni arendusprotsessi stabiilsuse kohta. Kui teenuse taastamise aeg on pikk, võib põhjuseks olla probleemid vigade tuvastamisel ja tagasisidel.

Kõige tähtsam ja ka keerulisem on defineerida intsident ja viis selle tuvastamiseks. Intsident on subjektiivne mõiste ning selle tüüpe võib olla mitmeid, ehk universaalse automatiseeritud lahenduse loomine on keerukas. Intsidendid võivad tekkida valest seadistusest või mõne sõltuvuse rikkest – need on sageli hallatud ja tuvastatud SLA mõõtmise tulemusena. Esineb ka funktsionaalseid vigu, mis SLA mõõtmisega tihti välja ei tule ja mille parandamise võimekuse mõõtmine on DevOps kontekstis oluline. Funktionaalsete vigade korral avatakse üldjuhul manuaalselt veapilet näiteks Jira keskkonnas. Funktionaalseid vigu saab tuvastada ka versioniseerimise järgi.

Mõõtmiseks võib kasutada intsidendi märkimise süsteemi ja selle oleku lõpetamist (näiteks Jira pileti kinnipanek) või parandavate paigalduste ehk kiirparanduse ja tagasipööramise tüüpi paigalduste (ingl *hotfix*/*rollback*) märkimist ja tuvastamist, ehk aeg vigasest paigaldusest kuni parandava paigalduseni. [14]

3.3.1 Võimalikud lahendused

Eksisteerib Jira keskkonnal põhinevaid olukorra identifitseerimise lahendusi, kus intsident tüüpi Jira objektide pealt arvutatakse selle avamise ja sulgemise vaheline aeg [14] [16]. On olemas ka Azure Monitoril põhinevaid lahendusi, kus Azure jälgib HTTP

veakoode ja rakenduse ressursside kasutust ning nende tekkel ja lahendamisel tuvastatakse vea teke ja süsteemi taastamise aeg [15]. Võimalik on tuvastada vea tekkeid ka manuaalse märkimise teel või kiirparanduse mustri avastamise teel Git haru nimes, liitumise päringu pealkirjas või *Deployment* objekti nimes [17].

3.3.2 Loodav lahendus

Eesmärk on luua Kubernetese platvormil põhinev võimalikult automatiseeritud ja lihtsasti paigaldatav rakendus. Manuaalse märkimise asemel on soovitud luua automaatne intsidendi tuvastus. Koodihoidla ja tööprotsessi piletite märkimise süsteemi integreerimine rakendusse on lõputöö skoopist väljas, sest lõputöö ajalise piirangu tõttu ei ole realistlik sellise ulatusega arenduse ja analüüsi teostamine. Lisaks, kõrvaliste monitoorivate tarkvarade kasutamisele ei soovita lahenduses toetuda.

Lahenduses kasutatakse klastrisse paigaldatud Docker tõmmiste versioonide semantilist muutumist, millega saab tuvastada funktsionaalseid vigu. Lahendus baseerub kiirparanduse ja tagasipööramise tüüpi paigalduste tuvastamisel. Pärast paigalduse sündmuse ja vastava *Event* objekti tekkimist kontrollitakse eelneva ja uue tõmmise versiooni ning kui need mõlemad on versioniseeritud SemVer mustri järgi, ehk versioonipõhine võrdlus on teostatav, kasutatakse järgmist algoritmi.

1. Kui tõmmise versiooni X.Y.Z mustri Z ehk *patch* versioon on suurem võrreldes eelneva klastris eksisteeriva sama *Deployment*-i tõmmise versiooniga, on tegu kiirparanduse tüüpi paigaldusega. Kiirparanduse tüüpi paigalduste puhul loetakse teenuse taastamise ajaks paigalduse sündmuse tekke hetke ja X.Y.0 mustri ehk sama *major* ja *minor* versiooni esimese paigalduse tekke ajavahemikku. Seda eeldades, et kõrvaldatavad vead tekivad alati esmase *minor* paigaldusega (X.Y.0), ning iga järgnev *patch* versioon ehk kiirparandus kõrvaldab algselt tekkinud probleemi ning *patch* versiooni suurendav paigaldus uusi vigu eraldi juurde ei tekita.

2. Kui tõmmise versiooni X.Y.Z mustri Z ehk *patch* versioon on väiksem (aga X ja Y on võrdsed) võrreldes eelneva klastris eksisteeriva sama *Deployment*-i tõmmise versiooniga, on tegu tagasipööramisega (ingl *rollback*). Seega eeldatakse, et tagasipööramisele eelnenud Docker tõmmis on alati vigases olekus. Teenuse

taastamise ajaks on tekkinud tagasipöörava paigalduse toimumise ajaline vahe võrreldes eelneva ehk vigase paigalduse tekke ajaga.

Kui muudetud *Deployment*-i eelneval või uuel tõmmisel pole Semver versioniseerimise muster tuvastatav, ehk versioonipõhine võrdlus pole teostatav, kasutatakse algoritmi:

1. Kui uue Docker tõmmise loomise hetk on vanem kui eelnev, on tegu tagasipööramise tüüpi paigaldusega. Jällegi, on teenuse taastamise ajaks tekkinud tagasipööramise paigalduse ajahetke vahe võrreldes eelneva (vigase) paigalduse tekke ajaga.

2. Kui uue Docker tõmmise loomise hetk on uuem eelnevast, kontrollitakse, kas uus tõmmis on Semver versioniseerimise mustriga. Kui on, kontrollitakse *patch* versiooni ja kui see on erinev nullist, on tegu kiirparanduse tüüpi paigaldusega. Teenuse taastamise ajaks on uue paigalduse tekke hetke ja X.Y.0 mustriga paigalduse hetke ajaline vahe.

3. Kui eelnevad kontrollid ei läbinud, on tegu uuenduse tüüpi paigaldusega.

3.4 Muudatuste ebaõnnestumise määr

Koos muudatuste jõustumise aja mõõdikuga hindab muudatuste ebaõnnestumise määra mõõdik arendusprotsessi stabiilsust. Kõrge muudatuste ebaõnnestumise määr juhib tähelepanu kesisele testimisele ja koodi ülevaatusele. Võimalikke lahendusi mõõdiku arvutamiseks on mitmeid:

- paigalduste protsent, millele järgnesid parandavad paigaldused;
- koodifikseeringute (ingl *commit*) arv, mille sõnumis/kirjelduses tuvastatakse kiirparanduse (ingl *hotfix*) märge;
- monitooringu tuvastatud intsidendid jagatud kõikide paigaldustega;
- manuaalne paigalduse staatuse määramine;
- tagasipööramiste arv jagatud kõikide paigaldustega. [14]

Kindlaks tuleb teha kaks juhtumit – mis on muudatus ja mis on ebaõnnestunud paigaldus. Muudatuseks on piisav pidada igasugust paigaldust. Ebaõnnestunuks saab pidada kõiki paigaldusi, millega seoses järgnesid mingisugused parandused.

3.4.1 Võimalikud lahendused

On lahendus, mis kasutab manuaalset paigalduse vigaseks märkimist läbi Jenkins keskkonna [14], või vastupidi, kus manuaalselt märgitakse paigaldus õigeks, ehk algselt on iga paigaldus “vigases“ olekus [15]. Nagu ka “Muudatuste ebaõnnestumise määr“ peatükis, saab ebaõnnestunud muudatust tuvastada ka nimemustrite ja monitoorimise programmide järgi. Seejärel tuleb mõõdiku arvutamiseks arvutada ebaõnnestunud paigalduste protsent kõikidest paigaldustest.

3.4.2 Loodav lahendus

Eesmärk on luua võimalikult automatiseeritult töötav rakendus. Ei saa loota manuaalse märkimise järjepidevusele inimlikest omadustest tulenevalt. Koodihoidla koodifikseeringute sõnumeid ei jälgita, sest selline lahendus nõuab Git integratsiooni ja liigset usaldust arendaja korrektses sõnumite kirjutamises. Lisaks arvatakse, et vea tuvastamiseks siinkohal ülesande piletite märkimise vahendit (näiteks Jira) kasutada ei saa, sest mida efektiivsem ja tootlikum arendusmeeskond, seda hetkelisemalt parandused tehakse, ilma midagi maha märkimata [14].

Saab toetuda juba välja mõeldud paigalduste tuvastamisele ja tüüpide määramisele. Muudatuste ebaõnnestumise määra mõõdiku arvutamine tugineb järkjärgulise toimunud paigalduste vigaseks määramisel. Iga parandav paigaldus jätab eelnevale juba esinenud paigaldusele märke külge, et paigaldust on parandatud.

Iga kord kui tekib kiirparanduse esimene versioon (näiteks X.Y.1), määratakse selle *minor* versiooni paigaldus (X.Y.0) vigaseks. Taoliselt määratakse ka kiirparanduse ja uuenduse tüüpi paigaldusi vigaseks, kui tekib tagasipööramine paigaldusest (näiteks tagasipööramine 1.0.3 versioonilt 1.0.2 versioonile märgib 1.0.3 versiooniga paigalduse vigaseks).

Sellisel viisil on mõõdiku arvutamine lihtne – tuleb võtta kõik uuenduse ja kiirparanduse tüüpi paigaldused ja jagada need vigaste paigaldustega. Tagasipööramise tüüpi paigaldusi siin ei arvestata, sest need ei too sisse uusi muudatusi, vaid võtavad juba esinenud koodi kasutusele.

4 Realisatsioon

4.1 Golang

Golang on avatud lähtekoodiga, staatiliselt tüübitud, kompileeritud programmeerimiskeel, mis on loodud Google poolt. Golang on ehitatud mitmetuumaliste protsessorite jaoks, sest funktsioone on võimalik jooksutada samaaegselt [18]. Golang pakub virtuaalmasinateta ja pausidetta prügikoristust (ingl *garbage collection*) ja programmikoodi kompileerimist masinkoodi [19].

Otsus kasutada Golang programmeerimiskeelt loodud rakenduses põhineb ka lahenduses kasutatavate ja sõltuvate tehnoloogiate ja platvormide jaoks kasutatud programmeerimiskeelest. Nimelt Kubernetes ja Docker on kirjutatud Golang-is, seega vajaminevate raamistike ja teekide arvukus on suur.

4.2 Andmemudel

Andmemudel on tähe skeemi ehk *star schema* tüüpi, mis on üks andmete modelleerimise tehnika andmeladudes. Andmeladu on kogum andmeid, mis on mõeldud analüütiliseks töötlemiseks OLAP tehnoloogiatega ning mida kasutatakse organisatsiooniliste otsuste tegemiseks. DevOps mõõdikute arvutamisel on samuti tegemist analüütilise töötlemisega ja tihtipeale tehakse seda väga suure koguse andmete pealt. Andmeladu hoitakse üldjuhul operatiivsest transaktsioonilisest andmebaasist (OTLP) lahus. Seda põhjusel, et andmelao disain peab toetama detailset analüütikat ning seda operatiivsed andmebaasid ei võimalda. Andmelao puhul on ajaloolised ja kokkuvõtlikud andmed tähtsamad kui individuaalsed kirjed. Kõige olulisem on siinkohal päringute kiirus. [20]

Tähe skeem koosneb faktidest ja dimensioonidest. Faktidesse kogutakse kõik eraldiseisvad huvipakkuvad sündmused. Dimensioonides paiknevad kõik süsteemsed objektid või muutujad, mis iseloomustavad fakti. Tähe skeem ei ole normaliseeritud, ehk tabelites esineb korduvaid väärtusi ja veerge. See aitab kaasa päringute kiirusele, ohverdades andmete liiasuse ja duplikatsiooni mitte esinemise. [21]

Loodud andemudel (Lisa) võimaldab tulemusi paindlikult ja detailselt esitada. Andmemudelis on kaks fakti tabelit – *fact_deployment*, mis kogub endasse kõik

tuvastatud paigaldused ja nende muudatuste jõustumise aja väärtuse (*lead_time_in_minutes*) ja *fact_restore* tabel, mis sisaldab endas kõiki paranduse tüüpi paigaldusi (kiirparandus, tagasipööramine) ja nende teenuse taastamise aja väärtust (*time_to_restore_in_minutes*).

Esineb viis dimensiooni tabelit:

1. *Dim_application* sisaldab endas esinenud Docker tõmmise nime (ilma versiooni märketa). Tõmmise nimi võib olla tõmmiste registri eesliidesega ja porti täpsustusega. Kui registri eesliidest täpsustatud pole, kasutatakse avalikku Docker registrit.

2. *Dim_version* kogub endasse kõik esinenud tõmmise versioonid, ning on mitu-ühele suhtes dim_applicationiga, sest ühe tõmmise nime kohta hakkab tõenäoliselt kogunema mitmeid erinevaid versioone. Lisaks versioonimärkele kogutakse tabelisse ka tõmmise räsi, mis erinevalt versioonimärkest (mis on puhtalt inimloetav alias) garanteerib tõmmise reaalse sisulise muutuse. Lisaks, on tabelisse kogutud veel tõmmise versioonist tulenevad *major*, *minor* ja *patch* väärtused, mis kogutakse juhul, kui on kasutatud Semver versioniseerimist.

3. *Dim_environment* tabelisse märgitakse maha Kubernetese klastri API serveri URL ja nimeruum. Kuberneteses ei ole klastrite eristamiseks otsest lahendust - nime klastrile määrata ei saa, seega kasutatakse klastri identifitseerimiseks Kubernetese API serveri URLi. Nimeruum märgitakse maha, et hiljem tulemuste agregeerimisel filtreerida vajadusel paigaldusi vastavalt nimeruumile (näiteks „dev“, „prod“).

4. *Dim_deployment_type* määrab paigalduse tüübi. Siia tabelisse jooksvalt kirjed ei looda – tabeli loomisel defineeritakse siia kolm kirjet ehk paigalduse tüüpi, milleks on uuendus, kiirparandus ja tagasipööramine.

5. *Dim_restore_type* määrab taastava paigalduse tüübi. Samuti siia tabelisse jooksvalt kirjeid ei sisestata, vaid kirjed on eelnevalt lisatud ja nendeks on kiirparandus ja tagasipööramine.

Loogiliselt on andmemudeli faktid ja paigaldused jagatud viisil, kus kõik uuenduse tüüpi paigaldused lähevad *fact_deployment* tabelisse. Kiirparanduse tüüpi paigaldused lähevad

nii *fact_deployment* tabelisse kui ka *fact_restore* tabelisse, sest kiirparanduse puhul on vajalik mõõta paigalduse muudatuste jõustumise aeg kui ka teenuse taastamise aeg. Tagasipööramise tüüpi paigaldust oleks esialgsel hinnangul vaja sisestada ainult *fact_restore* tabelisse, sest *fact_deployment* tabelis esinevat muudatuste jõustumise aega mõõta pole vajalik, kuna väärtus on minevikus juba arvutatud. Sellele vaatamata, on individuaalsest soovist tingitult tagasipööramise paigaldused ka *fact_deploymenti* sisestatud, kus *lead_time_in_minutes* veeru väärtuseks on teenuse taastamise aeg ja seda selleks, et hiljem tulemuste näitamisel oleks võimalikult lihtne tagasipööramise tüüpi paigaldusi kuvada ilma, et peaks keerukamaid UNION päringud tegema.

Rakenduse käivitamisel on eeldatud, et PostgreSQL andmebaas on kliendi enda poolt loodud, ja vajalikud ühendusparameetrid on rakenduse käivitamisel programmi argumentidega kaasa antud. Seejärel loob rakendus vajalikud tabelid andmebaasi ja hakkab paigaldusi ja kõike vajalikku jooksvalt andmebaasi sisestama.

4.3 Programmi argumendid ja keskkonnamuutujad

Programmi käivitades tuleb kaasa anda *„start”* argument ja seejärel programmi argumendid või keskkonnamuutujad (Tabel 1), milleks on nimeruumid (*namespaces*), andmebaasi *port* (*dbport*), andmebaasi võõrustaja (*dbhost*), andmebaasi kasutaja (*dbuser),* andmebaasi parool (*dbpassword*) ja andmebaasi nimi (*dbname*).

Tabel 1. Programmi argumendid ja keskkonnamuutujad

Muutuja	**Väärtuse näide**
namespaces	default,test,prod
dbhost	localhost
dbport	5432
dbuser	postgres
dbpassword	password
dbname	postgres
Näide - go run main.go start --namespaces=’prod,test' --dbhost='localhost' --dbport=5432 --dbuser='postgres' --dbpassword='password' --dbname='postgres'	

Nimeruumide muutujaga saab täpsustada, milliste Kubernetese klastri nimeruumide kohta infot koguda. Loodud rakendus filtreerib sündmused välja ainult märgitud nimeruumide kohta. Kui muutujat ei leita, kogutakse infot kõikide nimeruumide kohta.

Ülejäänud muutujad on vajalikud andmebaasiga ühenduse loomiseks. On eeldatud, et andmebaas on rakenduse kasutaja poolt loodud. Seejärel loob rakendus ühenduse, loob vajadusel vajaminevad tabelid andmebaasi ja hakkab tabelitesse infot sisestama.

Muutujate lahenduses on kasutatud Urfave [22] raamistikku, mis võimaldab mugavalt luua käsurea rakendusi Golang programmeerimiskeeles, kus rakenduse töötamiseks vajaminevate muutujate kogumine käib läbi keskkonnamuutujate ja rakenduse käivitamisel kaasa antud programmi argumentide.

4.4 Kubernetese klastri kuulamine

Rakendus pidevalt kuulab Kubernetese klastris toimuvaid sündmusi. Suhtlus klastriga on realiseeritav läbi Kubernetese API serveri. Läbi API saab pärida infot ja kontrollida klastri objekte (kaunasi, *Deployment* ja *Event* objekte).

4.4.1 Client-go

Rakenduses kasutatakse klastri kuulamiseks Client-go teeki, mis võimaldab Golang programmeerimiskeeles lihtsal moel Kubernetese klastriga suhelda ja selle sündmuseid jälgida. Kestva kuulamise jaoks on kasutatud teegi *Informer pattern* funktsionaalsust, mis võimaldab hoida järjepidevat ühendust klastri ja selle ressurssidega. *Informer*-iga kogutakse Kubernetese objekte vahemällu ja lugemise korral pöördutakse sama vahemälu poole. Selline lahendusviis võimaldab hoida pidevat ühendust klastri ressurssidega ja vähendab koormat Kubernetese API serverile ja ETCD-le, sest vajalik info on lokaalselt kättesaadav [23].

Informer-id on ressursipõhised ehk Kubernetese objekti põhised (näiteks *PodInformer* ehk kauna informeerija) ja need delegeerivad andmeid sündmuse käitlejatele (ingl *event handler*), pärast mida tehakse toiminguid vastavalt sündmuse tüübi järgi, milleks on lisamine, muutmine või kustutamine. Rakenduses on loodud *informer*-id kahele Kubernetese ressursile, *Deployment*-le ja *Event*-le (Joonis 1):

```
factory := informers.NewSharedInformerFactory(clientset, 0)
deployInformer := factory.Apps().V1().Deployments().Informer()
eventInformer := factory.Core().V1().Events().Informer()
synced = false
deployInformer.AddEventHandler(cache.ResourceEventHandlerFuncs{
   AddFunc:    onAddDeploy,
   UpdateFunc: onUpdateDeploy,
})
eventInformer.AddEventHandler(cache.ResourceEventHandlerFuncs{
   AddFunc: onAddEvent,
})
```

Joonis 1. Informeerijate ja sündmuse käitlejate loomine

Ressursse ja nendega seonduvaid sündmusi saab koguda vastavalt nimeruumile. *NewSharedInformerFactoryWithOptions* funktsiooniga saab kaasa anda nimeruumi parameetri, kus täpsustada saab ühe nimeruumi. Loodud lahenduses on soovitud koguda infot vajadusel mitme nimeruumi kohta (täpsustatud programmi argumentides), seega tuleb nimeruumi filtreerimine teha sündmuse käitlejates, kus kontrollitakse loodud või muudetud Kubernetese objektis täpsustatud nimeruumi. See aga toob sisse probleeme mälukasutusega, sest lokaalsesse vahemällu kogutakse teegi poolt infot ka ebaoluliste nimeruumide kohta, kuid pilootkatse kontekstis saab selle piiranguga arvestada ja keerukamat lahendust pole vaja.

Informeerija loeb käivitades kõik jälgitavad ressursid vahemällu. See tähendab, et programmi käivitades tekib iga klastris eksisteeriva ressursi kohta lisamise (*add*) sündmus. Neid sündmusi pole rakenduses vaja koguda. Seega tuleb rakendusel pärast informeerija käivitamist *cache.WaitForCacheSync* funktsiooniga lasta oodata kuni lokaalne vahemälu on täielikult täis laetud. Enne sündmuse käitleja jooksutamist rakendus kontrollib, kas vahemälu sünkroniseerimine on tehtud.

Informeerijat deklareerides tuleb parameetrina kaasa anda uuesti sünkroniseerimise perioodi väärtus (ingl *resync period*). See väärtus märgistab aja, kus võrreldakse vahemälus olevaid objekte reaalselt klastris eksisteerivate objektidega. Üldjuhul on need kogumid identsed, kuid võib juhtuda vääratusi erinevates komponentides, mistõttu jääb sündmus klastris (näiteks *Deployment* objekti muutumine) rakenduse poolt märkamata. Loodud rakenduses on uuesti sünkroniseerimise perioodiks määratud 0, mis tähendab, et uuesti sünkroniseerimist ei toimugi. Seda põhjusel, et huvi all on ainult jooksvad sündmused ning tõenäoliselt on rakenduse poolt tuvastatavad paigalduse sündmused väga sagedased ja pikaajalise mõõtmise tulemust mõne paigalduse kadumine ei mõjuta.

Client-go teek võimaldab kasutada ka dünaamilist informerit. See on kasulik juhul, kui pole alati teada, milliste Kubernetese objektidega tegutsetakse, näiteks CRD ehk kohandatud ressurssi (ingl *custom resource definition*) puhul. CRD on ressurss, mis on kasutaja enda poolt loodud ja ei eksisteeri Kuberneteses vaikimisi. Lõputöös käsitletava probleemi lahendamiseks pole dünaamilist informeerijat vaja, sest jälgitavad objektid on teada ja eksisteerivad klastris ka vaikimisi. [24]

4.5 Paigalduse sündmuse tuvastamine

Paigalduse sündmuse tuvastamine on jaotatud kahte etappi ehk vaja on tuvastada kahte Kubernetese klastris toimuvat sündmust. Esimeseks sündmuseks on klastri *Deployment* objekti konteinerite tõmmise versiooni muutus ja teiseks on vastava tõmmise alla laadimise sündmus klastris ehk *Event* objekti teke.

Paigalduse sündmuse tuvastamine põhineb Client-go teegi *Deployment*-ide informeerijal, mis tuvastab kõiksugu muutuseid klastri *Deployment*-ides. Kui muutus tuvastatud, kontrollitakse sündmuse käitlejas *Deployment* objekti nimeruumi vastavust programmi argumentides välja toodud nimeruumidele. Seejärel võrreldakse vana ja uue *Deployment* objekti konteinerite tõmmiseid ning iga muutunud konteineri kohta tekib eraldi paigalduse sündmus.

Pärast paigalduse tuvastamist luuakse *DeploymentEventObject* objekt, kuhu kogutakse kogu vajalik info, mida hiljem on vaja pärast Docker tõmmise alla laadimise sündmust ehk *Event*-i kasutada. Objekti kogutakse näiteks *Deployment*-i nimi, tõmmis, nimeruum ja klastri nimi (internetiaadress). Lisaks sellele, salvestatakse objekti *Deployment*-i kauna UID ehk unikaalne identifikaator. See on vajalik, et siduda tuvastatud paigalduse sündmus hiljem tekkiva tõmmise alla laadimise *Event*-iga, sest alla laadimise *Event* on seotud kaunaga, mitte *Deployment*-iga.

Selle jaoks otsitakse Client-go teegi abil välja kõik nimeruumis eksisteerivad kaunad ja tuvastatakse õige kaun nime järgi, sest kõik Kubernetese kaunad, mis kuuluvad *Deployment*-i alla, on nimeliselt alati *Deployment*-i nime eesliidestusega. Kui nimeline kontroll läbib, kontrollitakse ka kaunas sisalduva Docker tõmmise vastavust *Event* objekti kirjelduses tuvastatud tõmmisega. Seda põhjusel, et kontrollimise hetkel eksisteerib klastris kaks erinevat kauna, mis on nimeliselt sama eesliidestusega – üks neist on uus

värskelt tekkinud kaun, ja teine ehk vana kaun eksisteerib klastris kuni uue kauna tõmmis on alla laetud ja uus kaun on korrapäraselt töötamas.

Kui info kogutud ja objekt loodud, salvestatakse objekt lokaalsesse vahemällu, mis on realiseeritud *DeploymentEventObject* objektide massiivina nimega *CacheOfDeploymentEventObjects*. Kui järgmisena vastava Docker tõmmise alla tõmbamise *Event* tuvastatakse, seotakse massiivis olev *DeploymentEventObject* objekt *Event* objekti kauna UID ja tõmmise nimega, mis on välja toodud *Event* objekti kirjelduses. Massiivist õige objekti leidmiseks ootab programm ühe sekundi pärast igat ebaõnnestunud otsimise iteratsiooni, sest *DeploymentEventObject* objekti loomine võib teatud juhtudel kaua aega võtta (näiteks kauna UID-d otsides, kui nimeruumis eksisteerib tuhandeid kaunasid). Programm läbib viis iteratsiooni otsimiseks – kui õiget objekti ei leita, lõpetatakse protsess. Kui objektid seotud, on võimalik Docker tõmmise räsi ja loomise kuupäev leitud tõmmise metaandmetest pärida ja paigalduse sündmuse tüüp kindlaks määrata.

Saadud tõmmise räsi kontrollitakse eelnevalt eksisteerinud räsidega. Tõmmise räsi garanteerib tõmmise sisulise muutuse (tõmmise nimi ega versioon seda tegelikkuses ei garanteeri) [11]. Kui sisulist muutust pole toimunud, pole vaja uuesti muudatuste jõustumise aega mõõta, sest see on juba mõõdetud. Kui tegu on tagasipööramise tüüpi paigaldusega, on teenuse taastamise aeg vajalik siiski mõõta ja programm läbib oma tavapärast algoritmi. Vastasel juhul kustutatakse vastav paigalduse objekt vahemälust ja mõõtmisi ei teostata.

4.6 Paigalduse sündmuse tüübi määramine

Paigalduse sündmuse tüüpideks on uuendus, kiirparandus ja tagasipööramine ning selle tuvastamiseks on loodud kaks meetodit. *DetermineSemverEventType* (Joonis 2) funktsiooni rakendatakse juhul, kui eelneva ja ka uue tõmmise versioon on Semver mustriga, ehk versioonide vaheline võrdlus on teostatav.

```
func determineSemverEventType(deployObj dto.DeploymentEventObject)
string {
   oldTag := strings.Split(deployObj.OldImage, ":")[1]
   oldVersions := getSemverVersions(oldTag)
   oldVersionsAsSum := oldVersions["major"]*100 +
oldVersions["minor"]*10 + oldVersions["patch"]*1
   newVersionsAsSum := deployObj.DimVersion.MajorVersion*100 +
deployObj.DimVersion.MinorVersion*10 +
      deployObj.DimVersion.PatchVersion*1
   if newVersionsAsSum > oldVersionsAsSum {
      if deployObj.DimVersion.MajorVersion == oldVersions["major"] &&
         deployObj.DimVersion.MinorVersion == oldVersions["minor"] {
         if deployObj.DimVersion.PatchVersion > oldVersions["patch"] {
            return "HOTFIX"
         }
      } else {
         if deployObj.DimVersion.PatchVersion != 0 {
            return "HOTFIX"
         }
         return "ROLLFORWARD"
      }
   } else {
      return "ROLLBACK"
   }
   return "ROLLFORWARD"
}
```

Joonis 2. Paigalduse tüübi määramine Semver versioniseerimise abil

DetermineNonSemverEventType (Joonis 3) funktsiooni rakendatakse juhul, kui versioonide võrdlust rakendada ei saa, ehk kumbki tõmmise versioon ei ole Semver mustriga. Lahendus baseerub Docker tõmmiste loomise hetke võrdluses ning selleks kasutatakse ametlikku Docker-i avatud koodi, millega saab Docker Engine API-ga suhelda ja tõmmise loomise hetk metaandmetest välja lugeda.

```
func determineNonSemverEventType(deploymentEventObject
dto.DeploymentEventObject) string {
   if previousImageNewer(deploymentEventObject) {
      return "ROLLBACK"
   }
   if deploymentEventObject.NewImageSemverVersioned &&
      deploymentEventObject.DimVersion.PatchVersion != 0 {
      return "HOTFIX"
   }
   return "ROLLFORWARD"
}
```

Joonis 3. Paigalduse tüübi määramine tõmmise loomise aja abil

4.7 “Paigalduste sagedus“ mõõdiku arvutamine

„Paigalduste sagedus“ mõõdiku arvutamine on kõige lihtsam. Selleks loetakse kokku kõik paigaldused ehk andmebaasi tabeli *fact_deployment* kirjed. Paigalduse tüüp rolli ei mängi, ehk dimensiooni *dim_deployment_type* järgi paigaldusi filtreerima ei pea - arvutuse alla lähevad kõik uuenduse, kiirparanduse ja tagasipööramise tüüpi paigaldused. Seejärel saab tulemuse kokku agregeerida soovitud ajavahemiku jaoks, näiteks 1 nädal või 1 kuu.

4.8 “Muudatuste jõustumise aeg“ mõõdiku arvutamine

„Muudatuste jõustumise aeg“ mõõdikut arvutatakse uuenduse ja kiirparanduse tüüpi paigalduste puhul. Mõõdiku väärtuseks on tõmmise loomise ja paigalduse toimumise ajaline vahe ning tõmmise loomise aeg on päritav tõmmise metaandmetest ning on Golang programmeerimiskeeles kättesaadav läbi ametliku Docker teegi (Joonis 4).

```
func getLeadTimeInMinutes(image string) int {
   cli, err := client.NewClientWithOpts(client.FromEnv,
client.WithAPIVersionNegotiation())
   CheckError(err)
   defer cli.Close()

   imageData, _, err := cli.ImageInspectWithRaw(context.Background(),
image)
   createdAt, _ := time.Parse(time.RFC3339, imageData.Created)
   diff := time.Now().Sub(createdAt).Minutes()
   return int(diff)
}
```

Joonis 4. „Muudatuste jõustumise aeg“ mõõdiku arvutamine

Ühendus Docker-iga luuakse läbi *NewClientWithOpts* funktsiooni. Vajalikud parameetrid ühendamiseks tulenevad keskkonna muutujatest. Kui ühendus loodud, on võimalik käsurea liidest jäljendada ja tõmmise nime järgi metaandmed välja pärida.

4.9 “Teenuse taastamise aeg“ mõõdiku arvutamine

„Teenuse taastamise aeg“ mõõdikut mõõdetakse kiirparanduse ja tagasipööramise tüüpi paigalduste puhul. Taastamise ajaks on tagasipööramise puhul aeg eelneva ehk vigase paigalduse kasutuselevõtust kuni parandava paigalduse kasutuselevõtuni ja kiirparanduse puhul aeg intsidendi tekitavast paigaldusest ehk X.Y.0 versiooni kasutuselevõtust kuni parandava paigalduse kasutuselevõtuni. (Joonis 5).

```
func getTimeToRestoreService(restoreObj dto.RestoreEventObject) int {
   if restoreObj.DimRestoreType.RestoreType == "ROLLBACK" {
      lastDeploymentTimeAsString :=
dal.GetLastDeploymentTime(restoreObj, psqlconn)
      if lastDeploymentTimeAsString == "-1" {
         return -1
      }
      lastDeploymentTime, _ := time.Parse(time.RFC3339,
lastDeploymentTimeAsString)
      diff := time.Now().Sub(lastDeploymentTime).Minutes()
      return int(diff)
   } else { // HOTFIX
      minorReleaseTimeAsString :=
dal.GetMinorVersionReleaseTime(restoreObj, psqlconn)
      if minorReleaseTimeAsString == "NULL" {
         return -1
      }
      minorReleaseTime, _ := time.Parse(time.RFC3339,
minorReleaseTimeAsString)
      diff := time.Now().Sub(minorReleaseTime).Minutes()
      return int(diff)
   }
}
```

Joonis 5. „Teenuse taastamise aeg“ mõõdiku arvutamine

Tagasipööramise paigalduse sündmuse puhul on vajalik leida eelneva paigalduse toimumise aeg. See on leitav andmebaasist klastri API serveri aadressi, nimeruumi, *Deployment* objekti nime ning eelneva tõmmise nime ja versiooni järgi. Kui vastavat paigaldust andmebaasist ei leita siis uut paigaldust andmebaasi ei sisestata. Olukord võib tekkida, kui rakenduse käivitamine ja andmete kogumine algab pärast eelneva paigalduse toimumist.

Kiirparanduse paigalduse sündmuse puhul tuleb leida paranduste jada algus ehk versioon, kus *patch* on võrdne nulliga. Selle leidmiseks pöördutakse andmebaasi poole ja tulemus on leitav klastri API serveri aadressi, nimeruumi, paigalduse objekti nime, tõmmise nime ja *major* ning *minor* versiooni järgi. Kui parandavate paigalduste jada algust ei leita, ei ole teenuse taastamise aeg arvutatav ja uusi kirjeid andmebaasi ei sisestata. Selline olukord võib tekkida juhul, kui rakenduse käivitamine ja andmete kogumine saab alguse pärast vastava *minor* paigalduse teket.

4.10 “Muudatuste ebaõnnestumise sagedus“ mõõdiku arvutamine

Muudatuste ebaõnnestumise sageduse arvutamine põhineb toimunud paigalduste vigaseks määramisel. Iga kiirparanduse jada esmane versioon (X.Y.1) määrab paranduste

jada alguse (X.Y.0) vigaseks. Tagasipööramise puhul määratakse eelnev paigaldus alati vigaseks. Vigaseks määramine käib läbi *fact_deployment* tabeli *fix_required* välja.

Mõõdiku arvutamisel filtreeritakse kõik *fact_deployment* kirjed *dim_deployment_type* tabeli väärtuste ehk paigalduse sündmuse tüüpide järgi. Arvutamisel ei lähe arvesse tagasipööramise tüüpi paigaldused, sest need ei too kaasa koodi mõistes uusi muudatusi. Seejärel filtreeritakse kõik alles jäänud *fact_deployment* kirjed *fix_required* välja järgi – kirjete arv, kus *fix_required* on tõene, on muudatuste ebaõnnestumise sageduse mõõdiku väärtuseks.

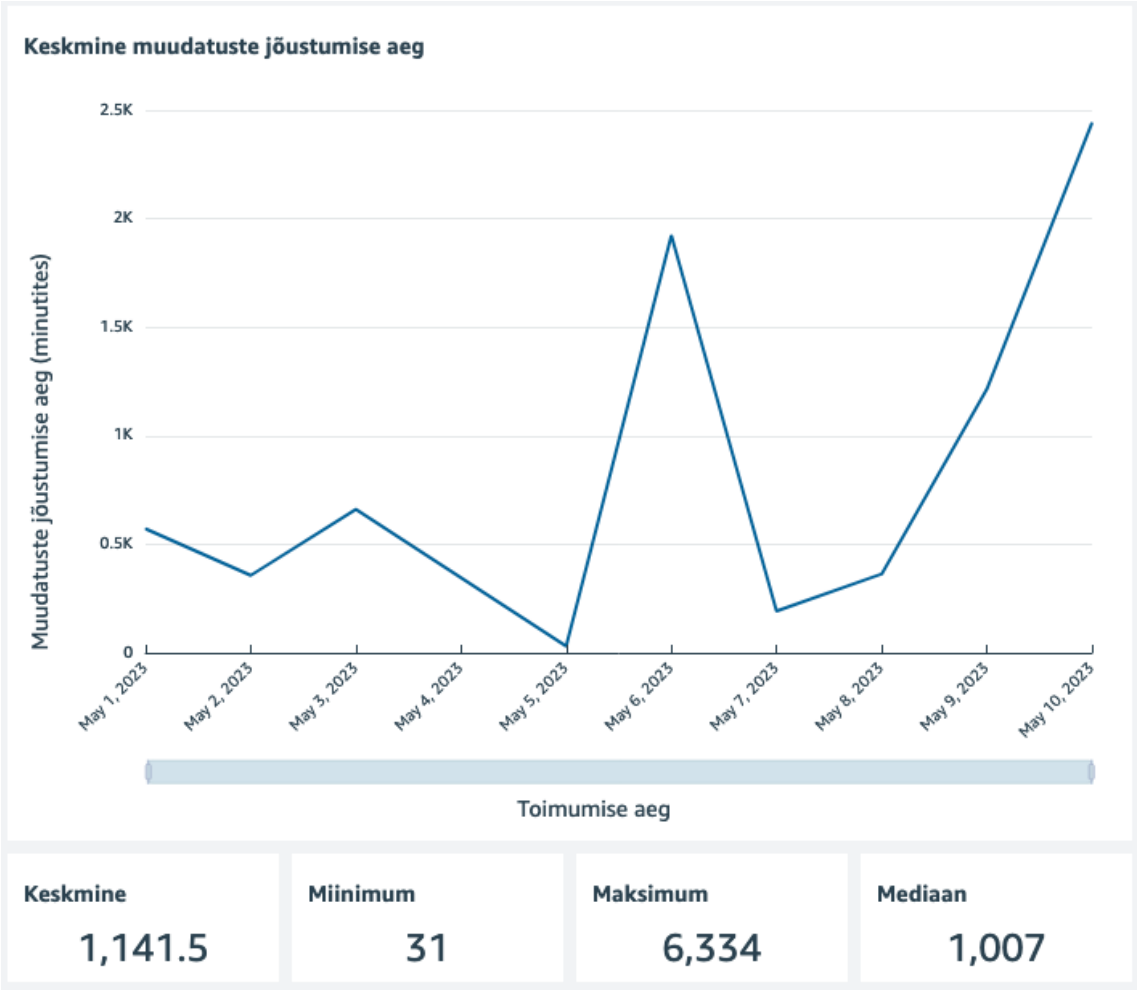
4.11 Tulemuste esitamine

Tulemuste esitamiseks on kasutatud Amazon Quicksighti, mis on 2015 aastal loodud ja käivitatud ärianalüütika teenus (ingl *business intelligence service*). Quicksight võimaldab ettevõtetel luua andmete visualiseerimisi ja analüüsida neid andmeid. Teenus kasutab kiiret mälusisest arvutusmootorit, mis töötab paralleelselt andmete arvutamiste tegemiseks. [25] Quicksight on skaleeritav, serverivaba (ingl *serverless*) ja masinõppel põhinev ning aitab ettevõtetel teha targemaid andmepõhiseid otsuseid. [26]

„Paigalduste sagedus“ mõõdikut arvutatakse vastavalt ajaperioodile – viimased 30 päeva, viimased 7 päeva ja tänane päev (Joonis 6).

Joonis 6. „Paigalduste sagedus“ Quicksight-is

„Muudatuste jõustumise aeg“ mõõdiku puhul kasutatakse visualiseerimiseks graafikut (Joonis 7). X-teljel on kuvatud kuupäevad ning Y-teljel muudatuste jõustumise aja keskmisi väärtusi. Lisaks on välja toodud terve kogumi peale keskmine, miinimum, maksimum ning mediaan väärtus.

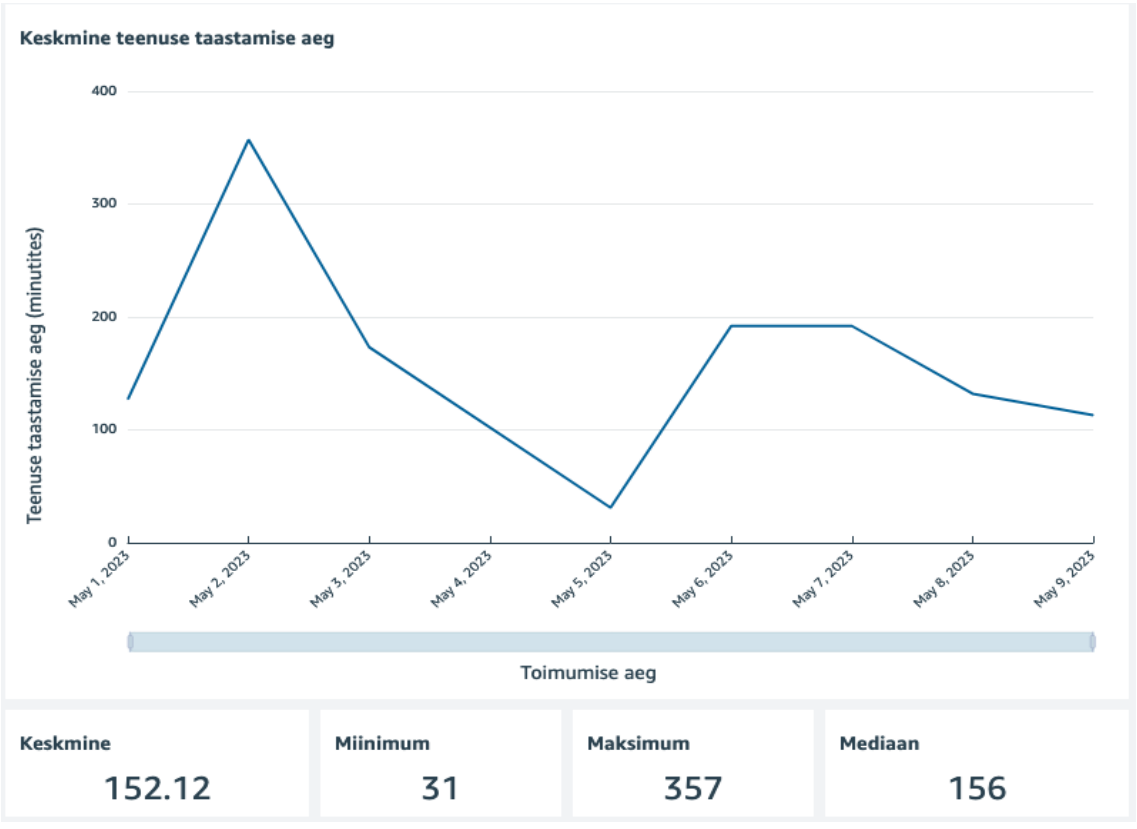


Joonis 7. „Muudatuste jõustumise aeg“ Quicksight-is

„Muudatuste ebaõnnestumine määr“ mõõdiku visualiseerimine (Joonis 8) on identne „paigalduste sagedus“ mõõdikuga – tulemused on kuvatud vastavalt ajaperioodile.

Joonis 8. „Muudatuste ebaõnnestumine määr“ Quicksight-is

„Teenuse taastamise aeg“ mõõdiku visualiseerimine (Joonis 9) on üks-ühele „muudatuste jõustumise aeg“ mõõdikuga. Kasutatud on graafikut ning välja on toodud ka tervest kogumist tulenev keskmine, miinimum, maksimum ning mediaan väärtus.



Joonis 9. „Teenuse taastamise aeg“ Quicksight-is

5 Edasine töö

Pilootkatse tulemusena on valminud rakenduse prototüüp, mille peale saab hakata jätkuarendusi looma. Prototüübi puhul on suurimaks puudujäägiks „muudatuste jõustumise aeg“ mõõdiku täpse arvutamise võimalikkus. Eesmärgi täieulatuslikuks täitmiseks on vaja integreerida rakendusse koodihoidlaga suhtlemine, et pärida täpne koodimuudatuste alustamise hetk koodifikseeringute abil.

Tööprotsessi piletite haldamise keskkonna (näiteks Jira) integreerimine aitaks muuta mõõdikute kogumist paindlikumaks – ei oleks tingimata vaja, et tõmmised oleksid Semver versioniseerimise mustriga, paigalduse tüübi määrab pileti tüüp ning pilet on seotud koodihoidla haru või paigaldusega näiteks nime järgi.

Ideaalse universaalse rakenduse loomine eesmärgi täitmiseks on keeruline, sest raske on luua töötavat rakendust ilma rangeid nõudeid rakenduse kasutajale seadmata. Nõudeid võib esineda koodihoidla ja tööprotsessi piletite platvormi määramisel, koodihoidla kasutamise strateegia määramisel (näiteks Git flow või *trunk-based*), tööprotsessi piletite ja koodihoidla harude või koodifikseeringute nimetamisel. Probleeme võib tekkida ka inimlikest vigadest tulenevate puudustega nõuete täitmisel.

Edasise tööna jätkatakse loodud rakenduse testimist ja oodatakse tagasisidet meeskondadelt ja juhtidelt, kas mõõdikute kasutamises nähakse väärtust. Läbi selle tekib arusaam, kas ideesse ja rakendusse on mõistlik investeerida ja kas edasiseid töid jätkata.

6 Kokkuvõte

Eesmärgiks oli luua rakendus, mis analüüsiks Kubernetese platvormil paikneva süsteemi arendusprotsessi, mille tulemusena saadakse väärtused vastavalt DevOps mõõdikutele. DevOps mõõdikud kirjeldavad arendava organisatsiooni või meeskonna võimekust edukalt tarnida, mis omakorda loob eeldused tootlikkuseks.

Lahendusena loodi rakendus, mis paikneb analüüsitavas Kubernetese klastris eraldi kaunana ja jälgib *Deployment*-idega seotud muutusi ja sündmuse objektide tekkimist. Mõõdikute kogumine baseerub paigalduse sündmuse identifitseerimisel ja selle tüübi määramisel. Paigalduse sündmus identifitseeritakse klastris paiknevate tõmmise muutuste järgi ning soovitatud on kasutada täpseks paigalduse tüübi määramiseks Semver versioniseerimise mustrit tõmmise versiooni määramisel. Kui Semver versioniseerimist teostada ei saa, võrdleb rakendus tõmmiste loomise aegu, et teha kindlaks paigalduse tüüp. Vastavalt paigalduse tüübile arvutatakse mõõdikuid ja sisestatakse vajaminevad andmed andmebaasi.

Rakenduse prototüüp, millega valideeritakse jätkuarenduste mõistlikkus ning mille peale saab jätkuarendusi teostada valmis edukalt. Rakendus täidab eesmärgid, kuid seab Semver versioniseerimise nõude rakenduse kasutajale. Võimalikke alternatiivseid lahendusi analüüsides selgub, et täpset ja universaalset lahendust, kus erilisi nõudeid kasutajale ei seata on keerukas luua. Kergem on luua kohandatud lahendusi. Täpsemate mõõtmistulemuste saamiseks on vajalik rakendusse integreerida koodihoidla ja tööprotsesside piletite süsteemi keskkondadega suhtlus andmete kogumiseks.

Rakendust testiti lokaalselt ning saadud tulemuste põhjal visualiseeriti andmed analüüsimiseks Amazon Quicksightis.

Kasutatud kirjandus

[1] N. Forsgren, D. Smith, J. Humble ja J. Frazelle, „2019 Accelerate State of DevOps Report,“ Google Cloud's DevOps Research and Assessment (DORA), 2019.

[2] C. Peters, D. Farley, D. Villalba ja D. Stanke, „2022 Accelerate State of DevOps Report,“ Google Cloud's DevOps Research and Assessment (DORA), 2022.

[3] C. Ebert, G. Gallardo, J. Hernantes ja N. Serrano, „DevOps,“ *IEEE Software,* kd. 33, nr 3, pp. 94-100, 2016.

[4] M. Skelton, M. Pais ja R. Malan, Team Topologies: Organizing Business and Technology Teams for Fast Flow, Oregon: IT Revolution, 2019.

[5] N. Forsgren, J. Humble ja G. Kim, Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations, Oregon: IT Revolution, 2018.

[6] B. Burns, J. Beda, K. Hightower ja L. Evenson, „Chapter 1. Introduction,“ %1 *Kubernetes: Up and Running*, Sebastopol, California, O’Reilly Media, Inc, 2022, p. 328.

[7] S. Dhadve, „[Latest] Global Kubernetes Market Size/Share Worth USD 7.8 Billion by 2030 at an 23.40% CAGR: Markets N Research (Share, Trends, Cap, Adoption, Forecast, Segmentation, Growth, Value),“ Markets N Research, 6 03 2023. [Võrgumaterjal]. Available: https://finance.yahoo.com/news/latest-global-kubernetes-market-size-153000315.html. [Kasutatud 1 04 2023].

[8] CNCF, „Kubernetes.io,“ Cloud Native Computing Foundation (CNCF), 24 10 2022. [Võrgumaterjal]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Kasutatud 10 04 2023].

[9] CNCF, „Kubernetes.io,“ Cloud Native Computing Foundation (CNCF), 18 02 2023. [Võrgumaterjal]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Kasutatud 10 04 2023].

[10] „What is Docker?,“ IBM, [Võrgumaterjal]. Available: https://www.ibm.com/topics/docker. [Kasutatud 1 May 2023].

[11] N. Poulton, Docker Deep Dive, Packt Publishing, 2018.

[12] „Semantic Versioning 2.0.0,“ [Võrgumaterjal]. Available: https://semver.org/. [Kasutatud 12 May 2023].

[13] N. Forsgren ja M. Kersten, „DevOps metrics,“ *Communications of the ACM,* kd. 61, nr 4, pp. 44-48, 2018.

[14] P. Gregory, C. Lassenius, X. Wang ja P. Kruchten, „Agile Processes in Software Engineering and Extreme Programming,“ %1 *Springer*, 2021.

[15] Sam Smith, „Samsmithnz/DevOpsMetrics,“ 12 04 2023. [Võrgumaterjal]. Available: https://github.com/samsmithnz/DevOpsMetrics. [Kasutatud 14 04 2023].

[16] „DORA - Median Time to Restore Service,“ Apache DevLake, [Võrgumaterjal]. Available: https://devlake.apache.org/docs/Metrics/MTTR. [Kasutatud 14 04 2023].

[17] E. Dundar, „DORA Metrics Tracking: How to Effectively Detect Production Failures,“ Oobeya, 30 01 2023. [Võrgumaterjal]. Available: https://oobeya.io/blog/dora-metrics-tracking-how-to-effectively-detect-production-failures/. [Kasutatud 14 04 2023].

[18] A. Sahu, „Turing,“ 24 4 2023. [Võrgumaterjal]. Available: https://www.turing.com/blog/golang-vs-java-which-language-is-best. [Kasutatud 24 04 2023].

[19] M. Andrawos ja M. Helmich, Cloud Native programming with Golang, Packt, 2017.

[20] S. Chaudhuri ja U. Dayal, „An overview of data warehousing and OLAP technology,“ *ACM SIGMOD Record,* kd. 26, nr 1, pp. 65-74, 1997.

[21] Databricks, „Databricks,“ Databricks Inc, [Võrgumaterjal]. Available: https://www.databricks.com/glossary/star-schema. [Kasutatud 24 04 2023].

[22] „urfave/cli,“ [Võrgumaterjal]. Available: https://github.com/urfave/cli. [Kasutatud 15 May 2023].

[23] S. Lai, „Explore client-go Informer Patterns,“ CodeX, 3 February 2022. [Võrgumaterjal]. Available: https://medium.com/codex/explore-client-go-informer-patterns-4415bb5f1fbd. [Kasutatud 8 May 2023].

[24] „Devpress,“ 08 04 2022. [Võrgumaterjal]. Available: https://devpress.csdn.net/k8s/62eb877f20df032da732b636.html. [Kasutatud 24 04 2023].

[25] M. Rahman ja H. Hasan, „Serverless Architecture for Big Data Analytics,“ %1 *2019 Global Conference for Advancement in Technology (GCAT)*, Bangalore, India, 2019.

[26] R. Awati, „Amazon QuickSight,“ TechTarget, June 2021. [Võrgumaterjal]. Available: https://www.techtarget.com/searchaws/definition/Amazon-QuickSight. [Kasutatud 9 May 2023].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Andree Kuriks

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "DevOps mõõdikute kogumine Kubernetese platvormil", mille juhendaja on Rein Remmel
   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.

15.05.2023

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.

Lisa 2 – Andmemudel