TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Martin Kokk 175861IAAB

Rakenduse katkestuseta konteinertehnoloogiale üleviimine ettevõtte näitel

Bakalaureusetöö

Juhendaja: Lauri Anton
BSc

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: Martin Kokk

15.05.2023

Annotatsioon

Diplomitöö eesmärgiks on üle viia virtuaalmasinal töötav rakendus katkestuseta konteinertehnoloogiale, kasutades selleks ettevõttes kasutusel olevat konteineriseerimise lahendust, konteinerite orkestreerijat ning pidevintegratsioon ja pidevvalmiduse tööriistu. Võrrelda koormusjaotureid, mis sobiks rakenduse sujuvaks üleviimiseks konteinerisse, saavutades sellega rakenduse jälgitavuse toimimise kui varasemalt kasutuses olev jälgitavuse tarkvara suletakse.

Töö tulemusel sai rakendus konteineriseeritud Kubernetese keskkonnas, kasutades rakenduse juurutamise pideintegratsioon ja pidevvalmiduse tööriistu. Valiti välja koormusjaotur HAProxy, mille juurutamisel saavutati rakenduse katkestuseta üleviimine virtuaalmasinalt konteinerisse, kasutades päringute suunamiseks HAProxy poolt pakutud funktsionaalsust suunata päringuid vastavalt seadistatud kaaludele.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 11 peatükki, 10 joonist, 2 tabelit.

Abstract

Migrating an Application to Container Technology Without Disruption Based on the Example of a Company

The aim of the thesis is to migrate a virtual machine application to container technology without disruption using an in-house containerization solution, a container orchestrator and a continuous integration and continuous delivery tools. To compare load balancers suitable for seamlessly migrating the application to a container, thereby achieving application observability when previously used observability software will shut down.

In the theoretical part, different container technologies are described, highlighting their advantages and disadvantages. An overview of the concept of continuous integration and continuous delivery and its tools. It explains what application observability means today and why it is necessary. It analyses the different load balancers that can be used to route requests towards the application in container, to achieve a seamless implementation of the application.

As a result of the work, the application was containerized in a Kubernetes environment, using the continuous integration and continuous delivery tools of the application deployment. The load balancer HAProxy was selected and deployed to achieve seamless migration of the application from the virtual machine to the container using the functionality provided by HAProxy to route requests according to the configured weights.

The thesis is in Estonian and contains 40 pages of text, 11 chapters, 10 figures, 2 tables.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface*, rakendusliides. Arvuti operatsioonisüsteemiga või rakendusprogrammiga määratud reeglistik, mille alusel rakendusprogramm kasutab operatsioonisüsteemi või teise rakendusprogrammi teenuseid.
A kirje	A kirje ehk aadressikirje omistab IP aadressile domeeninime.
CD	*Continuous Delivery*, pidevvalmidus. Pidevintegratsiooni edasiarendus, tarkvaraarenduse metoodika, mille järgi luuakse tarkvara lühikeste tsüklitena nii, et seda saab igal ajal (iga tehtud ja testitud muudatuse järel) väljastada.
CI	*Continuous Integration*, pidevintegratsioon. Ekstreemprogrammeerimisse kuuluv arendusmeetod (alates 1991) integratsiooniprobleemide kiireks väljaselgitamiseks komponentide tööeksemplaride sagedate (mitu korda päevas) automatiseeritud koosteoperatsioonidega.
CI/CD	*Continuous Integration and Continuous Delivery*, pideintegratsioon ja pidevalmidus
CNAME	*Canonical NAME*, kanooniline nimi. Reaalse domeeninime alias, millel on kõik originaali omadused, k.a. IP aadressid ja meilimarsruudid
DevOps	Kultuurifilosoofiate, tavade ja tööriistade kombinatsioon, mis suurendab organisatsiooni võimet pakkuda rakendusi ja teenuseid suure kiirusega: arendada ja parandada tooteid kiiremini kui traditsioonilisi tarkvaraarenduse ja infrastruktuuri haldamise protsesse kasutavad organisatsioonid.
DNS	*Domain Name System*, domeeninimede süsteem. Internetiteenus, mis tõlgib domeeninimed IP aadressideks.
Docker	Programm samanimeliselt firmalt, rakenduste ja nende sõltuvuste pakendamiseks porditavasse konteinerisse nende jaotamiseks ja juhtimiseks Linuxipõhises füüsilises või virtuaalkeskkonnas.
Dockerfile	Fail, mis koosneb Dockeri kujutiste koostamise juhistest.
HTTP	*HyperText Transfer Protocol*, hüperteksti edastuse protokoll. Rakenduskihi protokoll, veebi alus, on suunatud hüpermeediumile, määrab sõnumite vormingu ja edastuse, määrab veebiserverite ja brauserite käitumise.

Java arhiivfail	Fail, kuhu on kokku kogutud mingi Java apleti jaoks vajalikud klassi-, pildi- ja helifailid ning mis on seejärel kokku pakitud, ingl. *Java archive file*.
JDK	*Java Development Kit*, Java arenduskomplekt. Sellesse arenduskomplekti kuuluvad JVM, kompilaator, silur, jt. tööriistad Java aplettide ja rakenduste loomiseks.
Kaun	Ühest või mitmest konteinerist koosnev grupp, millel on ühised salvestus- ja võrguressursid ning mis omavahel moodustavad teenuse või komponendi, ingl. *pod*.
Kehtestama	Muudatusi püsivateks salvestama, näiteks koodihoidlas, ingl. *commit*.
Konveier	Järjestikku ühendatud andmetöötluselementide komplekt, kusjuures ühe elemendi väljund on järgmise sisendiks, ingl. *pipeline*.
K8s	Sõna Kubernetes lühend.
OCI	*Open Container Initiative*, avatud juhtimisstruktuur. Selle eesmärk on luua avatud tööstusstandardid konteinervormingute ja käitusaegade kohta.
Raamistik	Objektorienteeritud süsteemide puhul objektiklasside hulk, mis annab kasutajale või programmile omavahel seotud funktsioonide kollektsiooni, ingl. framework.
SVC	Kubernetese objekti *service* lühend.
Tooken	Objekt, mis tõendab õigust sooritada mingit toimingut, ingl. *token*.
Tõmmis	Öeldakse ka kui Dockeri tõmmis, on fail, mis sisaldab hulga juhiseid, mille täitmisel luuakse konteiner, ingl. *image*.
URL	*Uniform Resource Locator*, ühtne ressursilokaator. mehhanism ressursside identifitseerimiseks Internetis
VM	*Virtual Machine*, virtuaalmasin.
Üksuste testimine	Tarkvara- või riistvaraüksuste testimine. analüüsi- või teostusvigade avastamisek, ingl. *unit testing*.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Konteinertehnoloogiad on IT ettevõtetes aastatega aina rohkem populaarsust kogunud, millest tulenevalt pannakse suurem rõhk käivitada rakendusi konteinerites kui teada tuntud viisil virtuaalmasinates. Konteinerite kasutamine annab võimaluse rakenduste mugavamaks haldamiseks kui ka ressursi paremaks kasutuseks. Kaasaegses maailmas rakenduste traditsioonilised seire lahendused ei võimalda enam rakendusest täielikku ülevaadet saada ning selleks on vaja koos seirelahendusega kasutada ka rakenduste jälgitavuse lahendusi. Jälgitavus võimaldab ennetada vigu, mis rakenduses võivad esile tulla, kui ka juba teatud vigade puhul kindlaks, teha miks need esineda võivad.

Antud töös käsitletakse virtuaalmasinas töötava ärikriitilise rakenduse katkestuseta konteinertehnoloogiale üleviimist, tagamaks rakenduse jälgitavuse, kasutades selleks ettevõttes kasutusel olevaid konteinertehnoloogiaid ja pidevintegratsioon ja pidevvalmiduse töövahendeid. Valitakse välja sobivaim koormusjaotur, millega oleks võimalik rakenduse päringud katkestuseta suunata konteineris töötavale rakendusele.

Töö metoodilises osas kirjeldatakse erinevaid konteinertehnoloogiad, tuues välja nende eelised ja puudused. Antakse ülevaate pidevintegratsiooni ja pidevvalmiduse olemusest ja seda võimaldavatest tööriistades. Seletatakse, mida tähendab tänapäeval rakenduse jälgitavus ja miks see vajalik on. Analüüsitakse erinevaid koormusjaotureid, mis võimaldaksid teostada rakenduse poole tulevate päringute suunamist konteinerisse, et seeläbi saavutada rakenduse sujuv kasutuselevõtt.

Töö rakenduslikus osas käsitletakse rakenduse juurutamist konteinerisse. Tuuakse välja, mida on vaja rakenduse juures muuta või lisada, et rakendust oleks võimalik käivitada konteineris ja juurutada konteinerite orkestreerijasse. Näidatakse autori poolt välja valitud koormusjaoturi kasutamist rakenduse poole tulevate päringute suunamiseks.

Lõputöös kasutatud mõistete ja terminite tõlgeteks ja selgitusteks on kasutatud IT ja sidetehnika seletavat sõnaraamatut, Standardipõhist tarkvaratehnika sõnastikku ning Andmekaitse ja infoturbe leksikoni [1], [2], [3].

2 Probleem ja eesmärk

Ettevõttes on käimas jälgitavuse tarkvara vahetus seoses praegusel kasutusel oleva lahenduse Plumbr litsentsi lõppemisega. Seda hakkab asendama OpenTelemetry lahendus, mida on töö teostamise hetkel võimalik kasutusele võtta ainult konteinerites töötavatel rakendustel. Virtuaalmasinal töötavatel rakenduste puhul on antud hetkel kaks võimalust, kas viia rakendus konteinerisse või kaotada rakenduse jälgitavuse võimalus. Kuna tegemist on ärikriitilise rakendusega, mida siiani arendatakse, siis tuleb selle jälgitavus säilitada, et oleks võimaldatud vigade ning nendest tulenevate probleemide ennetamine ja hõlpsam lahendamine.

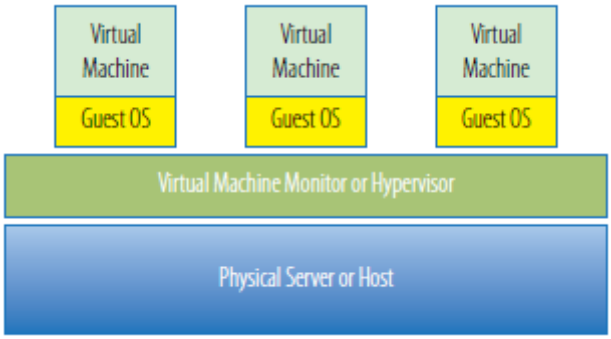
Rakenduse jälgitavus on tänapäeval aina aktuaalsemaks muutunud, kuna see võimaldab mõista süsteemi väljastpoolt, teadmata selle sisemist toimimist. Lisaks võimaldab jälgitavus hõlpsasti lahendada ja käsitleda uudseid probleeme. Sellest tulenevalt saab hinnata rakenduse edasisi parendusi arendusprotsessis.

Lõputöö eesmärgiks on üle viia virtuaalmasinas töötav rakendus katkestuseta konteinertehnoloogiale, kasutades selleks ettevõttes kasutusel olevat konteineriseerimise lahendust, konteinerite orkestreerijat, pidevintegratsioon ja pidevvalmiduse töövahendeid. Valida välja sobivaim koormusjaotur, et tagada rakenduse sujuv üleminek virtuaalmasinalt konteinerisse ilma katkestuseta. Sellega saavutada rakenduse jälgitavus ka siis, kui varasemalt kasutud lahendus enam ei tööta.

3 Virtualiseerimine

Virtualiseerimine viitab sageli mõne füüsilise komponendi lahutamisele loogiliseks objektiks. Objekti virtualiseerimisega saab objekti poolt pakutavast ressursist saada mingi suurema kasuliku meetme. Näiteks virtuaalsed kohtvõrgud (VLANid) pakuvad suuremat võrgu jõudlust ja paremat hallatavust, kuna need on eraldatud füüsilisest riistvarast. [4, lk. 2]

Virtuaalmasin (edaspidi VM) võib virtualiseerida kõik riistvararessursid, sealhulgas protsessorid, mälu, salvestusruumi ja võrguühenduse. *Virtual machine monitor* (VMM), mida tänapäeval nimetatakse tavaliselt hüperviisoriks, on tarkvara, mis pakub keskkonda, milles VM töötab. Täpsemalt on hüperviisor tarkvarakiht, mis asub virtuaalmasinate all ja riistvara kohal. Joonis 1 esitab virtuaalmasina loogilist skeemi. Ilma hüperviisorita suhtleb operatsioonisüsteem otse riistvaraga. Kettaoperatsioonid lähevad otse ketta allsüsteemi ja mälukutsed võetakse otse füüsilisest mälust. Ilma hüperviisorita tahab rohkem kui üks operatsioonisüsteem mitmest virtuaalmasinast samaaegselt kontrollida riistvara, mis omakorda tooks kaasa kaose. Hüperviisor haldab koostoimimist iga virtuaalmasina ja riistvara vahel, mida kõik virtuaalmasinad jagavad. [4, lk. 2, 22]



Joonis 1. Virtualiseerimise loogiline skeem [4].

3.1 Virtuaalmasin

Virtuaalmasinal on palju samu omadusi kui füüsilisel serveril. Nagu tegelik server, toetab virtuaalmasin operatsioonisüsteemi ja on konfigureeritud ressursikogumiga, millele

virtuaalmasinas töötavad rakendused saavad taotleda juurdepääsu. Erinevalt füüsilisest serverist, kus töötab korraga ainult üks operatsioonisüsteem ja mõned omavahel seotud rakendused, võib ühe füüsilise serveri sees töötada samaaegselt mitu virtuaalmasinat ning need virtuaalmasinad võivad kasutada palju erinevaid operatsioonisüsteeme, milles töötavad paljud erinevad rakendused. Erinevalt füüsilisest serverist ei ole VM tegelikult midagi muud kui kogum faile, mis kirjeldavad ja moodustavad virtuaalserveri. [4, lk. 37]

Peamised failid, millest VM koosneb, on konfiguratsioonifail ja virtuaalne kettafailid. Konfiguratsioonifail kirjeldab ressursse, mida VM saab kasutada: see loetleb virtuaalse riistvara, millest see konkreetne VM koosneb. Kui mõelda virtuaalmasinast kui tühjast serveri šassiist, siis konfiguratsioonifailis on loetletud, milliseid riistvaraseadmed selles šassiis oleksid: protsessor, muutmälu, salvestusruum, võrgundus, CD-ajam jne. Virtuaalmasinatel on juurdepääs erinevatele riistvararessurssidele, kuid nende seisukohast ei tea nad, et neid seadmeid tegelikult ei eksisteeri. Väljastpoolt virtuaalmasinat on näha vaid serveri ülesehitust, konfiguratsiooni ja süsteemiseadmeid. Virtuaalmasina sees on vaade identne füüsilise masina sees olemisega. Operatsioonisüsteemi või rakenduse seisukohalt on muutmälu, võrgundus ja töötlemine kõik kättesaadavad ja kõik näeb välja just nii nagu peabki. [4, lk. 37-39]

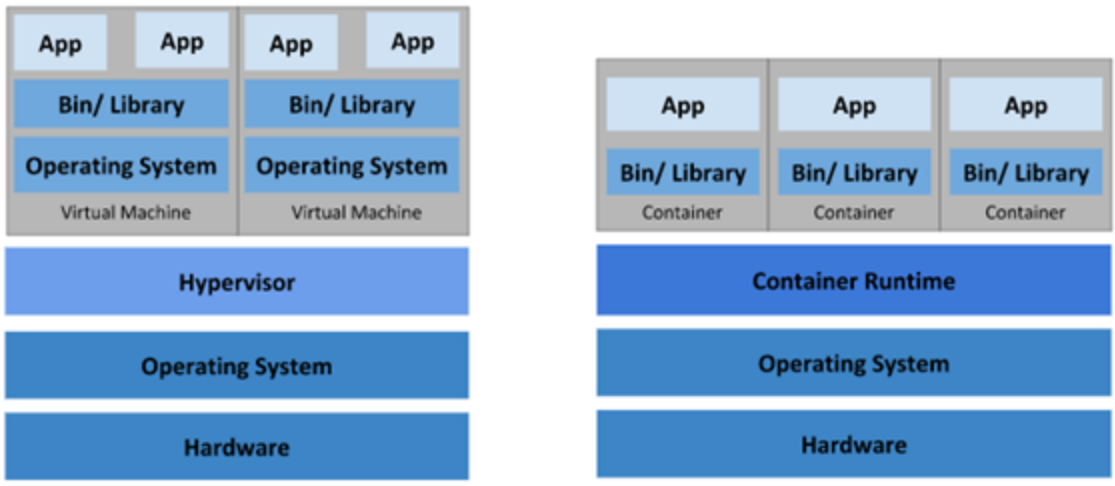
3.2 Konteinertehnoloogiad

Konteineriseermine sai populaarseks 2013. aastal koos Dockeri mootori kasutuselevõtuga. Konteinerite tehnoloogia on viimastel aastatel märkimisväärselt laienenud. Sajad organisatsioonid on rakenduste arendamise ja juurutamise kiirendamiseks kasutusele võtnud konteinertehnoloogiad. Huvi ja kasutuse kiire kasv konteineripõhiste lahenduste vastu on toonud kaasa vajaduse standardiseerimise järele. *Open Container Initiative* (edaspidi OCI), mille 2015. aasta juunis asutasid Docker ja teised tööstusharu juhid, edendab ühiseid minimaalseid avatud standardeid ja spetsifikatsioone konteinerite tehnoloogia kohta. Seetõttu aitab OCI laiendada avatud lähtekoodiga valikut. Kasutajad ei ole seotud konkreetse müüja tehnoloogiaga vaid pigem saavad nad ära kasutada OCI sertifikaadiga tehnoloogiaid, mis võimaldavad neil luua konteinerrakendusi, kasutades erinevaid DevOps tööriistade komplekte ja kasutada neid järjepidevalt enda valitud infrastruktuuril. [5], [6]

Konteiner on lihtsustatud, käivitatav tarkvaraüksus, mis pakib kokku rakenduse koodi ja sõltuvused nagu binaarkood, teegid ja konfiguratsioonifailid, et neid erinevates andmetöötluskeskkondades hõlpsasti juurutada [5]. Konteiner võimaldab isoleerida erinevad protsessid, mis töötavad hostil mingi virtualiseerimistehnoloogia abil, see võimaldab isoleerida konteinerid üksteisest ja konteinerid host-operatsioonisüsteemidest, võimaldades rakendusi igal pool samamoodi käivitada [7, lk. 296]. Lühidalt konteiner loob ja käitab rakendusi, eraldades konteinerid host operatsioonisüsteemist [5].

Vaatamata sellele, milline konteiner välja näeb, oma hostinime, failisüsteemi, protsesside ja võrgundusega, ei ole konteiner virtuaalmasin. Selles ei ole eraldi tuuma, seega ei saa tal olla eraldi tuumamooduleid või seadme draivereid. Konteineril võib olla mitu protsessi, kuid need peavad olema selgesõnaliselt käivitatud esimese protsessiga. Seega ei ole enamikul konteineritel ühtegi süsteemiteenust töötamas. [8, lk. 35-36]

Selleks, et käivitada rakendus virtuaalmasinas tuleb lisaks rakendusele ja selle sõltuvustele paigaldada kogu operatsioonisüsteem. Konteiner koosneb rakendusest endast ja tema sõltuvustest, näiteks teegid või teised rakendused [7, lk. 296]. Joonis 2 esitab virtuaalmasina ja konteineri erinevusi loogilisel skeemil.



Joonis 2. Virtuaalmasina (vasakul) ja konteineri (paremal) erinevused [9].

Tänapäeval on Docker üks tuntumaid ja laialdasemalt kasutatavaid konteinermootorite tehnoloogiaid, kuid see pole ainus saadaolev võimalus. Ökosüsteem standardiseerub konteineritel ja muudel alternatiividel, nagu CoreOS rkt, Mesos Containerizer, LXC Linux Containers, OpenVZ ja crio-d. Funktsioonid ja vaikeseaded võivad erineda, kuid

OCI spetsifikatsioonide kasutuselevõtt ja kasutamine nende arenedes tagab, et lahendused on tarnijaneutraalsed, sertifitseeritud töötama mitmes operatsioonisüsteemis ja kasutatavad mitmes keskkonnas. [6]

3.3 Konteinerite orkestreerimine

Konteinerprotsesside teisaldatavus ja reprodutseeritavus annab võimaluse liigutada ja skaleerida meie konteinerrakendusi üle pilvede ja andmekeskuste. Konteinerid tagavad tõhusalt, et need rakendused töötavad igal pool samamoodi, võimaldades meil kiiresti ja lihtsalt kõiki neid keskkondi ära kasutada. Lisaks, kui me oma rakendusi üles skaleerime, vajame mõningaid vahendeid, mis aitavad automatiseerida nende rakenduste hooldust, võimaldavad automaatselt asendada ebaõnnestunud konteinereid ning hallata nende konteinerite uuenduste ja ümberkonfiguratsioonide kasutuselevõttu nende elutsükli jooksul. Konteinerrakenduste haldamiseks, skaleerimiseks ja hooldamiseks mõeldud vahendeid nimetatakse orkestreerijateks. [10]

Konteinerite orkestreerimiskeskkond nagu Kubernetes võimaldab käsitleda mitut serverit ühe ressursikogumina, et käivitada konteinerid, jaotades konteinerid dünaamiliselt vabadele serveritele ja pakkudes hajutatud suhtlust ja salvestusruumi. Konteinerite orkestreerimiskeskkonna, kasutamiseks peab loobuma mõnest kontrollist konteinerite üle. Selle asemel, et täita käske otse konteinerite käivitamiseks, öeldakse orkesteerijale, milliseid konteinereid tahetakse käivitada ja see otsustab, kus iga konteinerit käivitada. Orkestreerija jälgib konteinereid ja tegeleb automaatse taaskäivituse, tõrkeülevaatuse, uuendustega ja isegi autoskaleerimisega koormuse alusel. [8, lk. 31, 37]

Samal ajal toob konteinerite orkestreerimiskeskkond kaasa ka väljakutseid mida ei pruugi olla olemas muud liiki rakendustaristute puhul. Kui konteiner on väikseim individuaalne moodul, mille ümber ehitatakse süsteem, siis rakenduskomponendid on palju isemajandavamad ja "läbipaistmatumad" infrastruktuuri seisukohast. See tähendab, et staatilise rakendusarhitektuuri asemel, mille kaudu valitakse, millised rakenduskomponendid on määratud konkreetsetele serveritele, püütakse orkestreerijaga teha võimalikuks, et iga konteiner töötaks kõikjal. [8, lk. 119]

Konteinerite elutsükli haldamiseks saab kasutada mitmeid konteineriorkestreerimise vahendeid. Mõned populaarsed võimalused on eelnevalt mainitud Kubernetes, kuid on ka

Docker Swarm ja Apache Mesos Marathon. Tänapäeval on Kubernetes kõige populaarsem konteinerite orkestreerimise platvorm ning enamik juhtivaid avalikke pilveteenuste pakkujaid nagu Amazon Web Services (AWS), Google Cloud Platform, IBM Cloud ja Microsoft Azure, pakuvad hallatavaid Kubernetese teenuseid. Tabel 1 toob välja eelnevalt mainitud orkestreerijate erinevused. [11], [12], [13]

Tabel 1. Konteineri orkestreerijate erinevused.

	Kubernetes	**Docker Swarm**	**Apache Mesos Marathon**
Lihtne kasutada	Keskmiselt keeruline	Lihtne	Keeruline
Klastri paigaldus	Keeruline	Lihtne	Keskmiselt keeruline
Minimaalne klastri suurus	1 peremees server ja 1 alluv server	1 peremees server	1 peremees server ja 1 alluv server
Konteineri juurutamine	YAML baasil konfiguratsioonifail	YAML baasil konfiguratsioonifail	JSON baasil konfiguratsioonifail
Sisemine ümbermarsruutimine	Iptables DNAT	IPVS	HAProxy

3.4 Konteinerite eelised virtuaalmasinatega võrreldes

Konteinereid võrreldakse sageli virtuaalmasinatega, sest mõlemad tehnoloogiad võimaldavad märkimisväärset jõudlust, võimaldades mitut tüüpi tarkvara, näiteks Linuxi- või Windowsi-põhist, ühes keskkonnas käivitada. Konteinertehnoloogia on aga tõestanud, et see pakub märkimisväärseid eeliseid üle virtualiseerimise ning on kiiresti muutumas IT-spetsialistide poolt eelistatud tehnoloogiaks. [6]

Konteinerite tehnoloogia üks suurepärane omadus on selle paindlikkus. Virtuaalmasinaga kulub käivitamiseks mitu minutit, kuid konteinertehnoloogia puhul saab konteineri käivitada mõne sekundiga, kuna operatsioonisüsteem on serveris juba käivitatud. See võimaldab konteinereid käivitada ja peatada vastavalt vajadusele, paindlikult käivitada tippnõudluse ajal ja paindlikult vähendada, kui seda ei ole vaja. [14]

Konteinerikeskkondades töötav tarkvara jagab masina operatsioonisüsteemi tuuma (ingl. *kernel*) ja konteineris olevaid rakenduskihti saab konteinerite vahel jagada. Seega on konteinerite võimsus olemuslikult väiksem kui virtuaalmasinal ja nad vajavad vähem käivitamisaega, mis võimaldab palju rohkem konteinereid käivitada sama jõudlusega kui

üks virtuaalmasin. See suurendab serveri tõhusust, vähendades serveri- ja litsentsikulusid. [6]

Kuna konteinerid pakivad mikroteenused ja nende sõltuvused väikesemahulisse paketti, on konteinereid lihtne liigutada, isegi erinevates keskkondades, näiteks avalikus pilves, privaatpilves ja hübriidpilves. Iga konteinerrakendus on isoleeritud ja töötab teistest sõltumatult. Ühe konteineri rike ei mõjuta teiste konteinerite jätkuvat tööd. Arendusmeeskonnad saavad tuvastada ja parandada kõik tehnilised probleemid ühes konteineris ilma teisi konteinereid mõjutamata. [15], [6]

Kuna enamus konteinereid sisaldavad tavaliselt vaid ühte rakendust, pakuvad nad lisaks väiksemat ründepinda võrreldes traditsioonilise rakendusserveriga. Enamikul rakendusserveritel töötab mitu taustateenust, sealhulgas suur hulk avatud porte. Konteinerites aga töötab üks isoleeritud programm, millel on tavaliselt vaid üks avatud port. Mõistlikud operatsiooniinsenerid saavad lukustada enamiku alussüsteemidest, millel konteinerid töötavad, jättes ainult piiratud pordid ja teenused, mis on vajalikud konteinerite töökoormuse nõuetekohaseks toimimiseks. [16]

3.5 Konteinerite puudused virtuaalmasinatega võrreldes

Konteinerid suurendavad rakenduste paindlikkust, kuid samal ajal lisavad mitmel viisil keerukust. Et konteineritega edukalt hakkama saada ja neile edukalt üle minna, tuleb kõigepealt tegeleda sellega seotud keerukustega. Keerukus võib tekkida turvalisuse, orkestreerimise, andmesalvestuse ja seire osas. [17]

Konteinersüsteemidele omane keerukus avaldub nii üksikute hostide tasandil kui ka kogu süsteemi arhitektuuris. Hosti tasandil lisab konteinermootori lisamine keerukust kohalikule võrgule, lisaks potentsiaalselt ebaturvaline deemon ja rakenduse jooksvate teenuste loetelule. Dockeri näitel sõltuvad konteinerid eraldi konteinerite alamvõrgust, mis töötab hostil. Insenerid peavad olema ettevaatlikud, et konfigureerida see võrk oma kasutusjuhtumi jaoks sobivalt, tagades, et peremees-süsteemiga jagatakse ainult minimaalne arv porte. [16]

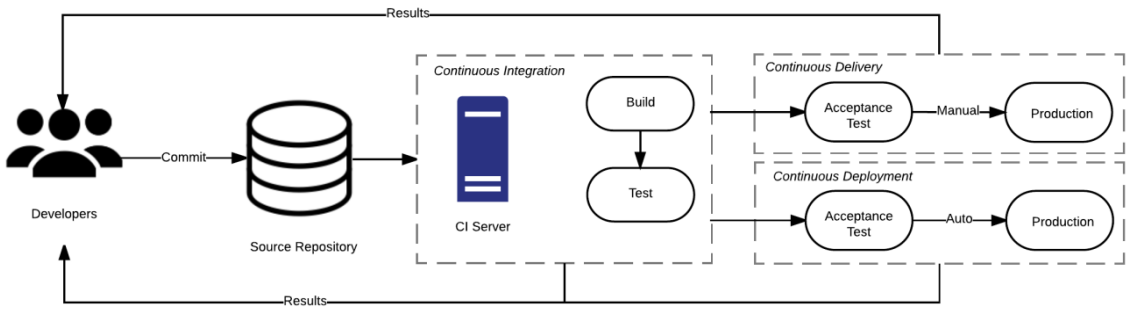
Vaadates süsteemi üldist arhitektuuri kõrgemal tasandil, võib konteinerite ja mikroteenuste lisandunud keerukus muuta ka loogilise turvalisuse keerulisemaks. Varasemates monoliitsetes rakenduste infrastruktuurides elasid nii rakendus kui ka

andmebaas ühes ja samas serveris, mis tähendab, et rakenduse eri osade vaheline suhtlus ei läbinud kunagi laiemat võrku. Kuid konteinerite mikroteenuste kasutuselevõtu puhul suhtlevad praktiliselt kõik rakenduse osad üle kogu võrgu, kusjuures üksikud API-d ja taustateenused on rakendatud konteinerites eraldi hostidel. Selleks, et suuremahuline mikroteenuste infrastruktuur oleks hallatav, on sageli vaja teenuste võrgustikku, mis omakorda tähendab, et teenuste võrgustiku turvalisus on elutähtis. [16]

Konteinerite seiresüsteemiga kaasnevad unikaalsed väljakutsed võrreldes traditsioonilise virtualiseeritud infrastruktuuri seirelahendusega. Esiteks konteinerid on lühiealised ehk neid saab kiiresti kasutusele võtta ja sama kiiresti hävitada. Selline käitumine on üks nende kasutamise peamisi eeliseid, kuid muudatuste jälgimine võib olla keeruline, eriti keerukates süsteemides. Teiseks konteinerid jagavad ressursse nagu mälu ja protsessor ühe või mitme konteineri vahel, mistõttu on raske jälgida ressursitarbimist füüsilisel serveril. See muudab konteinerite jõudluse kui ka rakenduse oleku kohta ülevaate saamise keeruliseks. Kolmandaks on ebapiisav tööriistade kasutamine ehk traditsioonilised seireplatvormid, mis sobivad ka virtualiseeritud keskkondade jaoks, ei pruugi anda piisavat ülevaadet meetrikatest ja logidest , mida on vaja konteinerite oleku ja jõudluse jälgimiseks kui ka vigade otsinguks. [18]

4 Pidevintegratsioon ja pidevvalmidus

Pidevintegratsioon ja pidevvalmidus (ingl. *continuous integration and continuous deilvery and continuous deployment*, edaspidi CI/CD) on meetod, mille abil saab rakendusi klientidele tarnida, võttes rakenduse arendamises kasutusele automatiseerimise. Pidevintegratsioon ja pidevvalmidusele omistatud põhimõisted on pidevintegratsioon, pidevvalmidus ja pidevjuurutamine. CI/CD on lahendus probleemidele, mida uue koodi integreerimine võib tekitada nii arendajatele kui ka administraatoritele. Täpsemalt, CI/CD toob kaasa pideva automatiseerimise ja pideva jälgimise kogu rakenduste elutsükli jooksul, alates integratsiooni- ja testimisfaasidest kuni tarnimise ja juurutamiseni. Neid omavahel seotud tavasid nimetatakse sageli CI/CD konveieriks (ingl. *pipeline*). Joonis 3 kujutab CI/CD protsessi. [19]



Joonis 3. Seos pidevintegratsiooni, pidevvalmiduse ja pidevjuurutamise vahel [20].

4.1 Pidevintegratsioon

Pidevintegratsioon (ingl. *continuoous integration*) on mitme osapoole koodimuudatuste integreerimise automatiseerimine ühte tarkvaraprojekti. See on peamine DevOps'i parim tava, mis võimaldab arendajatel sageli koodimuudatusi koondada kesksesse hoidlasse (ingl. *repository*), kus seejärel toimuvad rakenduse komponentide loomised ja testid. Uue koodi korrektsuse kinnitamiseks enne integreerimist kasutatakse automatiseeritud töövahendeid. [21]

Pidevintegratsioon aitab suurendada insenerimeeskondade töötajate arvu ja tarnetulemusi. Pidevintegratsiooni kasutuselevõtt võimaldab tarkvaraarendajatel töötada

funktsioonide kallal iseseisvalt ja paralleelselt. Kui nad on valmis neid funktsioone lõpptootesse ühendama, saavad nad seda teha iseseisvalt ja kiiresti. Pidevintegratsioon on väärtuslik ja väljakujunenud tava kaasaegsetes, suure jõudlusega tarkvaraarendusega tegelevates organisatsioonides. [21]

Pidevintegratsiooni abil teevad arendajad sageli kehtestusi (ingl. *commit*) ühises hoidlas, kasutades sellist versioonihaldussüsteemi nagu Git. Pidevintegratsiooni teenus koostab automaatselt uue koodi muudatused ja viib läbi üksuste testimise (ingl. *unit testing*), et tuua kohe esile kõik vead. Pidevintegratsioon käsitleb üldiselt rakenduse koostamise ja üksuste testimise etappe tarkvara avalikustamise protsessis. Iga kehtestatud versioon, mis edastatakse hoidlasse, käivitab automaatse koostamise ja testimise protsessid. [22]

4.2 Pidevvalmidus ja pidevjuurutamine

Pidevvalmidus (ingl. *continuous delivery*) on tarkvaraarenduse tava, mille puhul koodimuudatused valmistatakse automaatselt ette toodangukeskkonnas avaldamiseks. Pidevvalmidus on kaasaegsete rakenduste arendusprotsessi alustala, mis laiendab pidevintegratsiooni rakendades kõik koodimuudatused testkeskkonda ja/või tootmiskeskkonda pärast koostamisetappi. Õigesti rakendatuna on arendajatel alati juurutamisvalmis rakendus, mis on läbinud standardiseeritud testimisprotsessi. [23]

Pidevvalmiduse konveier (ingl. *pipeline*) võib olla manuaalne tegevus, mis nõuab inimese sekkumist, vahetult enne tootmiskeskkonda juurutamist. Arendusmeeskond hoiab pärast iga sprinti toote juurutamiskõlbliku versiooni valmis ja ärimeeskond teeb lõpliku otsuse toote vabastamiseks kõigile klientidele või elanikkonna läbilõikele või ehk inimestele, kes elavad teatavas geograafilises asukohas. [24]

Pidevjuurutamine (ingl. *continuous deployment*) läheb pidevvalmidusest veel ühe sammu võrra kaugemale. Selle tava puhul väljastatakse klientidele iga muudatus, mis läbib kõik tootmisprotsessi etapid. Inimese sekkumine puudub ja ainult ebaõnnestunud test takistab uue muudatuse tootmisse juurutamist. Pidevjuurutamine on suurepärane võimalus kiirendada tagasisideahelat oma klientidega ja võtta meeskonnalt surve maha. Arendajad saavad keskenduda tarkvara loomisele ja näevad, et nende töö läheb reaalajas käiku mõni minut pärast töö lõpetamist. [25]

4.3 Pidevintegratsioon ja pidevvalmiduse tööriistad

Tänapäeval on olemas palju suurepäraseid CI/CD tööriistu. Mõned on olnud turul juba pikka aega, teised on suhteliselt uued. On veidi üleliigne öelda, et kaasaegne CI-vahend peab olema kiire, kasutajasõbralik ja paindlik, sest need on omadused, mida juba eeldatakse, et on olemas. CI/CD vahendid võib jagada järgmistesse kolme põhikategooriasse [26, lk. 16]:

- Pilvepõhised lahendused, nagu AWS CodePipeline, Google Cloud Build ja Microsoft Azure Pipelines.
- Avatud lähtekoodiga lahendused nagu Jenkins, Spinnaker või GoCD.
- Tarkvara kui teenus (*software-as-a-service*, edaspidi SaaS) lahendused nagu Travis CI, CircleCI, TeamCity ja Bamboo.

Saadaval on palju suurepäraseid CI/CD vahendeid, seega peate valima parima, mis põhineb järgmistel teguritel [26, lk. 17-18]:

- Meeskonna kogemused ja oskused - kuigi paljud tööriistad kasutavad CI/CD-konveieri deklareerimiseks konfiguratsiooni YAML-faile, võivad nad nõuda mõningaid süsteemiadministraatori oskusi CI/CD-platvormi käivitamiseks vajaliku infrastruktuuri loomiseks ja pakkumiseks.
- Sihtplatvorm – ehk millisel operatsioonisüsteemil rakendus või projekt töötab (mõned CI-vahendid ei toeta macOS-i ja ARM-arhitektuuri) ning tööriist paigutatakse privaatpilve või avalikku pilve.
- Programmeerimiskeel ja arhitektuur - enamik CI-vahendeid toetab tipptasemel programmeerimiskeeli, sealhulgas Java, Ruby, Python, PHP ja JavaScript. Mõned tööriistad, nagu TeamCity, pakuvad siiski paremat integratsiooni ja tuge Java- ja .NET-projektidele. Sarnaselt on Atlassiani loodud Bambool tugi Jira ja Bitbucket töövahenditele. Lisaks võib projekti jaoks õige CI/CD vahendi valimisel olla määravaks teguriks ka juurutamislahendus. Tööriistad nagu Drone ja GitLab CI pakuvad Dockeri tuge koos integreeritud Dockeri registriga.

5 Rakenduse jälgitavus

Traditsiooniline lähenemine, mille kohaselt kasutatakse tarkvara mõõtmisi ja seiret, et aru saada, mida tarkvara teeb, jääb drastiliselt puudulikuks. Varem võis see tööstusharu hästi teenida, kuid kaasaegsed süsteemid nõuavad paremat metoodikat. Seire praktika põhineb paljudel süsteemidega seotud sõnastamata eeldustel arhitektuuri ja organisatsiooni kohta. Need eeldused on tavaliselt nähtamatud kuni need tekivad, misjärel need lakkavad olemast varjatud ning aitavad mõista, mis toimub. Eeldusteks on näiteks rakenduse töötamine virtuaalmasinas, mida saab kontrollida ning süsteemimõõdikute, milleks on näiteks muutmälu kasutatavus ja protsessori keskmine koormus, kättesaadavus. Kuid kuna süsteemid arenevad edasi, muutudes üha abstraktsemaks ja keerulisemaks ning nende aluseks olevad komponendid hakkavad üha vähem tähtsust omama, muutuvad need eeldused üha vähem tõepäraseks. Seetõttu traditsioonilised seiremeetodid on süsteemide mõistmiseks ebatõhusad ja kaasaegses maailmas ei toimi. [27, lk. 8-10]

Mõiste "jälgitavus" võeti kasutusele insener Rudolf E. Kálmáni poolt 1960. aastal. Oma 1960. aasta artiklis esitas Kálmán matemaatiliste kontrollisüsteemide kirjeldamiseks iseloomustuse, mida ta nimetas jälgitavuseks. Kontrolliteoorias määratletakse jälgitavust kui mõõdikut, mis näitab, kui hästi saab süsteemi sisemisi olekuid tuletada süsteemi väliste väljundite teadmisest. Lihtsustatult öeldes on tarkvarasüsteemides mõiste "jälgitavuse" mõõt, mis näitab, kui hästi saab mõista ja selgitada mis tahes seisundit, millesse süsteem võib sattuda, ükskõik kui uudne või kummaline see ka poleks. Jälgitavus võimaldab mõista süsteemi väljastpoolt teadmata selle sisemist toimimist, mille abil hõlpsasti lahendada ja käsitleda uudseid probleeme ning leida vastused küsimustele miks midagi juhtub. [27, lk. 4], [28]

Traditsioonilised seirevahendid töötavad, kontrollides süsteemi tingimusi võrreldes teadaolevate lävenditega, mis näitavad, kas esineb eelnevalt teadaolevaid vigu. See on põhiliselt reaktiivne lähenemisviis, sest see toimib hästi ainult varem esinenud vigade tuvastamisel. Seevastu jälgitavuse vahendid töötavad, võimaldades iteratiivset uurivat uurimist, et süstemaatiliselt kindlaks teha, kus ja miks võivad esineda jõudlusprobleemid. Jälgitavus võimaldab ennetavat lähenemist mis tahes vigade tuvastamiseks, olenemata

sellest, kas tegemist on eelnevalt teada või teadmata vigadega. Seire on mõeldud teadaolevate tundmatute probleemide jaoks, kuid jälgitavus on mõeldud teadmata tundmatute probleemide jaoks. [27, lk. 19], [27, lk. 13]

Varasemalt võis rakendustesse paigaldada teeke või agente, mis on tagarakenduse (ingl. *back-end*) spetsiifilised. Seejärel lisatakse kohandatud instrumente, et koguda mis tahes andmeid, mida peetakse rakenduste sisemise seisundi mõistmiseks oluliseks, kasutades funktsioone, mida klient teegid pakuvad. Kuid nende instrumenteerimise lähenemisviiside tootespetsiifiline olemus loob ka tootjapõhisuse, nagu näiteks Plumbr, Honeycomb ja Lightstep. Kui soovitakse saata telemeetriaandmeid teisele tootele, peab kogu instrumenteerimis protsessi kordama, kasutades teistsuguseid teeke, dubleerides raiskavalt koodi ja kahekordistades mõõtmise üldkulusid. Seire ja jälgitavuse kogukond on loonud mitmeid avatud lähtekoodiga projekte aastate jooksul, püüdes lahendada tootepõhisuse probleemi. Open-Tracing ja OpenCensus loodi vastavalt 2016. ja 2017. aastal. Need konkureerivad avatud standardid pakkusid teeke kõige populaarsematele programmeerimiskeeltele et võimaldada telemeetriaandmete kogumist, et neid saaks reaalajas edastada enda valitud taustsüsteemile. [27, lk. 74]

2019. aastal lõid Open-Tracing ja OpenCensus koos OpenTelemetry projekti. OpenTelemetry (edaspidi OTel) on nii OpenTracingi kui ka OpenCensus'i järgmine suurversioon, mis pakub täielikku asendust ja säilitab ühilduvuse. OTel salvestab jälgede, meetrikate, logide ja muude rakenduste telemeetriaandmeid ning võimaldab need saata enda valitud taustsüsteemile. OpenTelemetryst on saanud rakenduste instrumenteerimise avatud lähtekoodiga standard jälgitavuse lahenduste seas. OpenTelemetryga saab rakenduse koodi ainult üks kord instrumenteerida ja edastada telemeetriaandmed mis tahes valitud taustsüsteemi, olgu see siis avatud lähtekoodiga või patenteeritud. [27, lk. 74-75]

6 Hetkeseis ettevõttes

Antud peatükk räägib üldiselt rakendusest, kus rakendus praegu töötab, kuidas seda juurutatakse ning millisesse konteinerite orkestreerija keskkonda tuleb see üle viia.

6.1 Rakendus ja selle juurutamine

Rakenduse on üles ehitatud REST arhitektuuri põhimõtete järgi, mis kasutab tänapäeval tuntud Spring Boot raamistikku (ingl. *framework*) ja on kokku pakitud Java arhiivfailiks ning see on käivitatud virtuaalmasinas. Virtuaalmasin on paigaldatud ettevõtte andmekeskuses olevatel serveritel, mis on mõeldud ainult ühe rakenduse jaoks ehk sinna ei ole paigaldatud midagi muud peale operatsioonisüsteemi ja rakenduse jaoks vajalike tööriistade. Kuna rakendus kasutab Spring Boot raamistikku, siis on selle töötamiseks vajalik Java paketi olemasolu, millega saab rakenduse virtuaalmasinas tööle panna. Antud rakendus kasutab Java paketi versiooni OpenJDK 11.

Rakenduse juurutamist teostatakse pidevintegratsiooni ja pidevvalmiduse põhimõtteid järgides, antud ettevõttes on nendeks töövahenditeks Bitbucket ja Bamboo. Bitbucketi kui koodi hoidla on koht, kus asuvad kõik antud rakendusega seotud failid ja Bamboo puhul on tegemist tööriistaga, mis teostab rakenduse kokku panemise, testide tegemise kui ka juurutamise. Kuna Bambool otsest liidest virtuaalmasinaga ei ole, siis paigaldus skriptide käivitamiseks on kasutusel Ansible konfiguratsioonihaldus tööriist, mis teostab rakenduse tarne virtuaalmasinasse, koos tema vajalike sõltuvustega näiteks kindla Java versiooni pakett ja käivitab selle.

6.2 Konteinerite orekstreerija ettevõttes

Ettevõttes on kasutusel Kubernetese konteinerite orkestreerija tööriist. Kahele erinevale andmekeskusele on paigaldatud Kubernetese klaster, mille keskmeks on koormusjaotur (ingl. *load balancer*). See tähendab, et ühes klastris olev rakendus ei tea teises klastris olevast rakendusest midagi. Arhitektuur on loodud *active-active* põhimõtteid kasutatavatele rakendustele, kus klastrite vahel kasutatakse koormusjaoturit. Sellisel

juhul jagatakse koormust rakendusele klastrite vahel ning ühes klastris olev rakendus ei saa olla passiivne ehk ei võta tööd üle kui teises klastris oleva rakendusega peaks probleeme olema. Ettevõttes on iga rakenduse kohta tehtud eraldi nimeruum (ingl. *namespace*), mis grupeerib rakendusega seotud kaunad (ingl. *pods*), teenused, seadistuse ja paroolid. Nimeruumidele ligipääs on turvatud rollide põhiselt, kus igale kasutajale tekitatakse eraldi tooken (ingl. *token*), millega saab ligi ainult nendele nimeruumidele, kuhu kasutaja on lisatud.

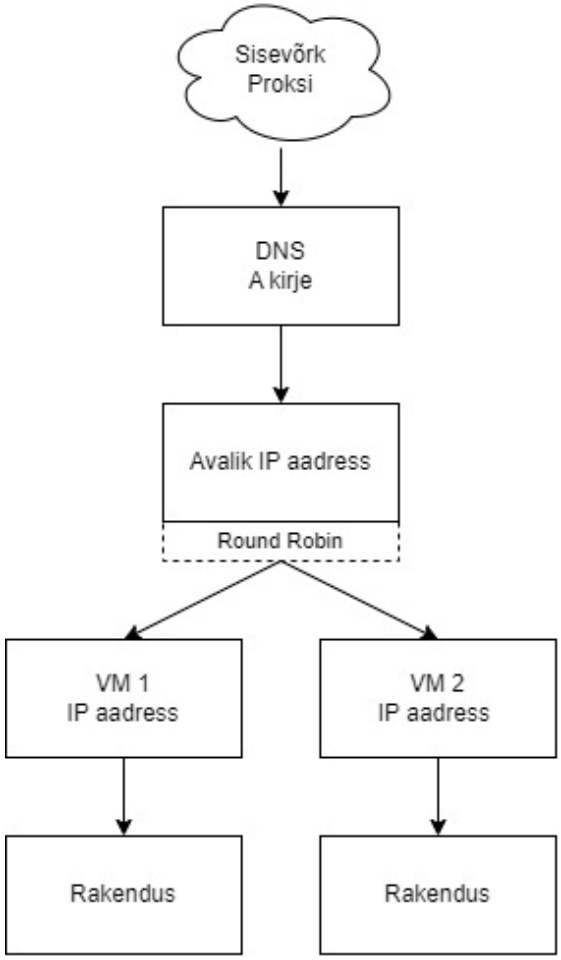
Kubernetesesse juurutamisel tuleb kasutusele võtta ka konteinerite register (ingl. container registry), mis on keskne koht, kuhu Dockeri tõmmiseid üles laetakse ja kust need tõmmised Kubernetese klastrisse alla laetakse, et käivitada rakendus uue tõmmisega. Seal hoiustatakse tõmmiste versioone, et uue rakenduse versiooni vea korral oleks võimalik vana versioon kiirelt, ilma uuesti rakendust ehitamata, tööle panna. Samuti peavad tõmmised alati olema kättesaadavad kui peaks Kubernetese kauna taaskäivituma, mis juhul tõmmatakse registrist sama tõmmis uuesti alla.

6.3 Rakenduse üleviimise keerulisus

Rakenduse üleviimise keerulisus seisneb selle töösoleku säilitamises kui rakendus viiakse üle konteinertehnoloogiale. Kuna rakendust pidevalt arendatakse, lisades sellesse uut funktsionaalsust. Selleks on oluline, et rakendust oleks võimalik igal ajahetkel juurutada erinevatesse keskkondadesse ja samuti oles ka kättesaadav kõikides keskkondades. Rakenduse ehitamine on üks osa juurutamise juures, kus koostatakse rakenduse JAR fail ning salvestatakse koodi hoidlast vajalikud failid Bamboo keskkonda, mida kasutatakse juurutamise juures. Rakenduse konteineriseerimiseks vajaliku tõmmise loomine lisab pidevintegratsiooni uue etapi, mis ei tohi algset töövoogu katkestada.

Samuti on rakendus seotud ka teiste süsteemidega, mis puhul peab rakendus päringutele vastama. Erinevate rakenduste teenuste kasutamiseks on ettevõttes loodud keskne proksi tööriist, kuhu on koondatud teenuste loetelu, mida siserakendused pakuvad ja mis on piiratud kasutajaõigustega. Iga rakenduse kohta on proksis ära kirjeldatud tema DNS aadress kuhu poole päringuid suunatakse. DNS aadress on seotud, kas mitmes virtuaalmasinas töötavate rakenduste IP aadressidega või Kubernetese klastris oleva rakenduse DNS nimega. Joonis 4 esitab rakenduse hetkeseisu, kuidas päringu rakenduseni jõuab. Antud rakenduse puhul on tegemist DNS A kirjega, millele on

omistatud avaliku IP aadress ning, mis on seotud kahe virtuaalmasina (VM) IP aadressiga, selliselt on tegemist kasutades *Round Robin* algoritmi koormuse tasakaalustamiseks. Hetkelahenduse korral ei liigu päringud konteineris töötavale rakendusele pärast selle konteineriseerimist.



Joonis 4. Päringute liikumise skeem enne koormusjaoturi lisamist.

Konteineriseeritud rakenduse keskkond asub täiesti teises alamvõrgus, mis on turvatud tulemüüriga, mille puhul on vaja iga rakenduse jaoks tekitada ligipääs läbi tulemüüri rakendusele. Selleks, et maandada riske ja inimeste tekitatud vigu, luuakse eraldi koormusjaotur rakenduse ette, mis suunab päringud nii rakendusele virtuaalmasinas kui ka konteineris. Samuti annab koormusjaotur oma algoritmiga muuta kaalu, kui palju päringuid mingis keskkonnas olevale rakendusele suunatakse. Nii saab suunata rohkem päringuid konteineris olevale rakendusele ning seejärel jälgida rakenduse tööd, mida võimaldab ettevõttes olev uus jälgitavuse tööriist.

7 Koormuse tasakaalustamine ja koormusjaoturite analüüs

Antud peatükis selgitatakse koormuse tasakaalustamise põhimõtetest, koormusjaoturi vajalikkusest antud töö probleemi lahendamisel ja seda võimaldavatest koormusjaoturitest. Seejärel toob autor välja tänapäeval kolm koormusjaoturit, võrdleb neid ja valib välja ühe, mis vastab kõige paremini etteantud tingimustele, et seda oma töös kasutada.

7.1 Koormuse tasakaalustamine

Koormuse tasakaalustamine tähendab võimet jaotada rakenduse töötlemiskoormus mitme eraldi süsteemi vahel, et suurendada sissetulevate päringute töötlemise üldist jõudlust. Koormuse tasakaalustamine kujutab endast võimet kanda mis tahes osa süsteemipäringute töötlemisest üle mõnele teisele sõltumatule süsteemile, mis seda samaaegselt töötleb. See jagab ühte serverisse saabuva koormuse mitme teise seadme vahel. Selle eeliseks on, et see vähendab esmase vastuvõtva serveri poolt tehtava töötlemise mahtu, mis võimaldab tal töödelda rohkem taotlusi ja suurendab jõudlust, kuna ressursside pärast on vähem konkurentsi ja kogu koormus on jaotatud rohkemate seadmete vahel. [29, lk. 109]

Koormuse tasakaalustamisega tegeleb vahend või rakendus, mida nimetatakse koormusjaoturiks. Koormusjaotur võib olla kas riistvarapõhine või tarkvarapõhine. Riistvaralised koormusjaoturid nõuavad spetsiaalse koormust tasakaalustava seadme paigaldamist, tarkvarapõhised koormusjaoturid võivad töötada serveris, virtuaalmasinas või pilves. Kui mõnest teisest serverist tuleb päring, määrab koormusjaotur päringu konkreetsele serverile ja see protsess kordub iga päringu puhul. Koormusjaoturid määravad mitmete erinevate algoritmide alusel kindlaks, milline server peaks iga päringuga tegelema. Need algoritmid jagunevad kahte põhikategooriasse: staatilised, dünaamilised ja loodusest inspireeritud. [30]

Staatilised koormuse tasakaalustamise algoritmid, kus staatilises keskkonnas koormuse tasakaalustamise algoritmide protsess sõltub eelnevatest teadmistest süsteemi olekust

koos selle omaduste ja võimalustega. Eelneva teabe hulka võivad kuuluda näiteks muutmälu, mälumaht ja töötlemisvõimsus. Selline teave kujutab endast süsteemi koormust. Staatilised algoritmid ei võta arvesse koormuse dünaamilisi muutusi töö ajal. Seega on nende algoritmide peamiseks puuduseks madal tõrketaluvus, mis tuleneb koormuse äkilistest muutustest. Näiteks algoritmid: *Round Robin, Weighted Round Robin* ja *Min-Min* [31]

Dünaamilised koormuse tasakaalustamise algoritmid on seda tüüpi algoritmid, mis on teadaolevalt paremad ja kohandatavad koormuse tasakaalustamiseks. Erinevalt staatilise koormuse tasakaalustamise algoritmidest võtavad koormuse tasakaalustamise algoritmid dünaamilises keskkonnas arvesse süsteemi varasemat seisundit. Seega on nende algoritmide peamine eelis paindlikkus, kuigi need võivad tunduda keerulised ja põhjustada süsteemi suurt koormust, seega peaksid selle kategooria uued kavandatud algoritmid selliseid puudusi vältima. Näiteks algoritmid: *Throttled, Equally Spread Current Execution, Least Connection* [31]

Loodusest inspireeritud koormuse tasakaalustamise algoritmid esindavad bioloogilisi protsesse või tegevusi, mis põhinevad inimloomusel, näiteks mesilaste geneetiline protsess mee leidmiseks. Neid protsesse modelleeritakse matemaatiliselt, et võtta koormuse tasakaalustamiseks kasutusele looduslikud protsessid. Nende intelligentsete algoritmide väljatöötamine toimib paremini keeruliste ja dünaamiliste süsteemide puhul. Näiteks algoritmid: *Honey-Bee, Ant Colony, Genetic, Particle Swarm* [31]

7.2 Koormusjaoturi vajalikkus

Rakenduse konteinerisse ja selle Kubernetesesse juurutamisest ainult ei piisa. Kuna tegemist on teise alamvõrguga ning kesksel proksil ei ole selle rakenduse kohta infot, et päringuid sinna suunata. Selleks, et proksi seadistust mitte muuta ja jätta rakendus korraga tööle nii virtuaalmasinas kui konteineris on vajalik võrguseadistust muuta. Algse lahenduse puhul, esitatud joonisel 4, on proksis seadistatud DNS aadress, mis vastas ühele avalikult IP aadressile ning see aadress vastas kahele virtuaalmasina IP aadressile, kuhu päringud edasi suunatakse.

Jäädes kasutama vana lahendust, ning lisada avaliku IP fondi veel kaks IP aadressi, mis vastaks Kubernetese kahe koormusjaoturi IP aadressile, mille kaudu saaks rakendusele

konteineris ligi, poleks võimalik päringuid vastavalt kaalule suunata kuna sellisel juhul kasutatakse ainult *Round Robin* algoritmi. Selleks, et oleks võimalik määrata rakenduse kaalusid, et suunata rohkem päringuid konteinerites olevatele rakendustele. See võimaldab testida rakenduse toimimist, et tuvastavad jälgitavuse abiga, kas rakenduse päringud jõuavad ka teistesse rakendustesse kohale, kas rakendus saab andmebaasile ligi ja kas rakenduses kuvab õigeid andmeid. Näiteks, kui tuleb välja, et rakendus ei saa teise rakenduse poole päringuid saata kuna tulemüüris puudub vastab kirje selles, siis on kiirem viis koormusjaoturis kaale muuta kui oodata, millal tulemüüri kirje lisatakse.

7.3 Koormusjaoturite analüüs

Käesolevas peatükis analüüsitakse populaarsemaid koormusjaotureid. Nende analüüsimisel valitakse sobivaim koormusjaoturi, mis vastab järgmistele tingimustele:

- Lähtekoodtarkvara
- Laialdaselt kasutatud
- Tarkvara jõudlus
- Lihtne kasutada
- Algoritmi *weighted round robin* kasutamise võimalus

Koormusjaoturite valimisel võeti arvesse Gartner grupi alla kuulava G2 portaali populaarsemate koormusjaoturite nimekiri, mis on mõeldud keskmise suurusega ettevõtetele. Sellest nimekirjast lähtudes on võetud kolm populaarsemat koormusjaoturit, mis pakuvad tarkvarapõhist koormuse tasakaalustamist ning mis ei nõua avaliku pilvelahenduse olemasolu [32]:

- Kemp LoadMaster
- HAProxy
- F5 Nginx

7.3.1 Lähtekoodtarkvara

Kemp LoadMaster ei ole lähtekoodtarkvara, tegemist on kommertsliku tootega, mis paigaldatakse hüperviisori kaudu virtuaalmasinasse. Kemp LoadMaster pakub ka tasuta tarkvara kui koormusjaoturi läbilaske võime ei ületa 20 megabitti sekundis [33]. Sellegipoolest ei kuulu Kemp LoadMaster lähtekoodtarkvara alla kuna tegemist ei ole avatud lähtekoodiga ning selleks, et pääseda ligi tasuta versioonile on vaja ennast nende kodulehel registreerida.

HAProxy kui ka F5 Nginx pakuvad mõlemad koormusjaotureid lähtekoodtarkvarana [34], [35]. Tarkvara on võimalik alla laadida HAProxy ja F5 Nginx hoidlatest, kuid on toetatud ka konteineritesse mõeldud tõmmised konteinerite registris Docker Hub. Selliselt on nende tarkvara võimalik paigaldada tasuta nii virtuaalmasinatesse kui ka konteineritesse.

7.3.2 Laialdaselt kasutatud

Kolmest koormusjaoturist on HAProxy ja F5 Nginx laialdaselt rohkem kasutusel kui Kemp LoadMaster. Kuna Kemp LoadMaster ei paku lähtekoodtarkvaralist lahendust kõigile kättesaamiseks. Seda kinnitab ka G2 portaali turupositsiooni graafik, kus F5 Nginx ja HaProxy on üle keskmise ja Kemp LoadMaster asub graafiku keskmisest allpool [32].

HAProxy kohta on ka öeldud, et tegemist on *de-facto* avatud lähtekoodiga koormuse tasakaalustaja, mida kasutatakse nii Linuxi distributsioonidel kui ka pilveplatvormidel. HAProxy on tuntud, kuna nende koormusjaotur on väga kiire ja usaldusväärne pöördproksi, mis pakub TCP- ja HTTP-põhiste rakenduste jaoks kõrget kättesaadavust, koormuse tasakaalustamist ja vahendamist [34]. Seega tegemist on spetsiaalselt koormuse tasakaalustamiseks mõeldud tarkvaraga, mis on 21 aastat turul olnud.

F5 Nginx on samuti avatud lähtekoodiga kuid mida teatakse põhilisel veebiserveri tarkvarana. Täpsemalt F5 Nginx on suure jõudlusega HTTP server, pöördproksi ja meiliserveri proksi server, mis on tuntud oma suure jõudluse, stabiilsuse, rikkalike funktsioonide, lihtsa seadistamise ja madala ressursikulu poolest [35]. Üks nendest funktsioonidest on ka koormuse tasakaalustamine, mida ei ole vaja eraldi lisada vaid on vaikimisi tarkvaras olemas. F5 Nginx loodi juba 2004. aastal kuid suurema tuntuse saavutas aastal 2011.

7.3.3 Tarkvara jõudlus

Kemp LoadMaster poolt pakutava koormusjaoturi lahenduse jõudlus on vastav koormusele kuid Kemp pakub tasulist koormusejaoturit, millel saab kergesti ressurssi juurde lisada kui koormus tõuseb. Sellest tulenevalt ei saa Kemp LoadMasterit võrrelda tasuta lähtekoodtarvaralisite lahendustega HAProxy ja F5 Nginx.

Võrreldes lahendusi HAProxy ja F5 Nginx on nende jõudlus pealtnäha samaväärsed kuna mõlema dokumentatsioonis on kirjas, et suudavad toime tulla suure koormusega. Alfred Johanssoni poolt koostatud lõputöös aastal 2022 viidi läbi uuring testimaks nelja koormusjaoturi jõudlus, millest kaks olid HAProxy ja F5 Nginx. Töös loodi kaks stsenaariumi, millest selgus, et kõige suurema koormuse korral saavutas HAProxy kõige parema jõudluse ning oli 36% parem võrreldes F5 Nginx koormusjaoturiga, mis omakorda saavutas kõige kehvema tulemuse [36]. Sellest järeldades on HAProxy parema jõudlusega ning stabiilsem koormujaotur kui F5 Nginx.

7.3.4 Lihtne kasutada

Kuigi Kemp LoadMaster pole nii laialdaselt kasutatud kui teised koormusjaoturid, siis tema eelised on kasutusmugavus, kasutajakogemus kui ka Kemp poolt pakutav tugi. Ehki koormusjaoturi paigaldus võib olla keeruline, siis koormusjaoturi seadistamine on tehtud võimalikult lihtsaks. Kemp LoadMaster pakub omaltpoolt palju erinevaid malle, millega seadistamist lihtsamaks muuta.

HAProxyt kui ka F5 Nginx tarkvara on kergem paigaldada kui Kemp LoadMasteri, näiteks kõige lihtsam viis neid paigaldada on kasutades Dockeri tõmmist ning käivitada see konteineris. Nii F5 Nginx kui HAProxy dokumentatsioon on lai ning kergesti kättesaadav, mis on kerge jälgitav. Kuid HAProxy seadistamine nõuab dokumentatsiooni rohkem süvenemist, et vastavalt vajadusele koormusjaoturit kasutada. See eest pakub HAProxy palju erinevaid funktsionaalsusi koormusjaoturi kasutamiseks, selleks on näiteks rakenduse töösoleku kontrollimine, mis F5 Nginx tarkvaral puudub. Sellest tulenevalt pakub HAProxy paremat tuge kasutajatele kui F5 Nginx. F5 Nginx seadistamine on kasutajatele võimalikult lihtsaks tehtud, et kasutajad saaksid vähese vaevaga rakenduse tööle.

7.3.5 Algoritmi *weighted round robin* kasutamise võimalus

Staatiline algoritm *weighted round robin* sai valitud antud töös, kuna see võimaldab kõige lihtsamalt koormust tasakaalustada, andes rakendustele kaalu, kuhu kõige rohkem päringuid suunata ning sellega saavutada rakenduse sujuv üleminek ühest keskkonnast teise. Kõige tavalisem koormuse tasakaalustamise algoritm *round robin* töötab ringikujulise ja korrastatud protseduuriga, kus igale protsessile on määratud kindel ajavahemik ilma igasuguse prioriteedita. Weighted round robin on sarnase lähenemisega nagu traditsiooniline round robin, kuid see algoritm arvestab iga sõlme (ingl. *node*) kaalu, mida tuleb eelnevalt käsitsi seadistada [31].

Kuna tegemist on laialt levinud staatilise algoritmiga, mis võimaldab suunata päringuid vastavalt sõlmedele ette antud kaaludele, siis on see enamast lisatud ühe algoritmina, mida koormusjaoturites kasutada. Sellest tulenevalt on algoritm olemas koormusjaoturitel Kemp LoadMaster, HAProxy kui ka F5 Nginx, mida kõik koormusjaoturid võimaldavad kasutada ning mis on vaikimisi tarkvaraga kaasas [37], [38], [39].

7.3.6 Koormusjaoturi valimine

Tingimuste võrdluste alusel loodi võrdlustabel (Tabel 2), kus iga tingimuse kohta anti vastavalt 0, 0,5 või 1 punkti.

Tabel 2. Koormusjaoturite võrdlus

Tingimus	**Kemp LoadMaster**	**HAProxy**	**F5 Nginx**
Lähtekoodtarkvara	0	1	1
Laialdaselt kasutatud	0,5	1	0,5
Tarkvara jõudlus	0,5	1	0,5
Lihtne kasutada	0,5	0,5	1
Algoritmi *weighted round robin* kasutamise võimalus	1	1	1
Kokku	2,5	4,5	4

Kemp LoadMaster on G2 portaali järgi populaarseim koormusjaotur pakkudes lihtsat seadistamist ja head jõudlust. Kuid tegemist ei ole lähtekoodtarkvaraga, millel oleks avatud lähtekood. Samuti on Kemp LoadMasterit raskem paigaldada kui teisi töös

kajastatud koormusjaotureid. Koormusjoaturi tugevused on Kemp poolt pakutav hea tugi kui ka staatiliste koormuse tasakaalustamise algoritmide olemasolu.

F5 Nginx on koormusjaoturite seas tuntud ja HAProxy kõige suurem konkurent turul, mis pakub tasuta lähtekoodtarkvara, kuid tegemist ei ole ainult koormuse tasakaalustamiseks mõeldud tarkvaraga. F5 Nginx loodi veebiserveri tarkvaraks, mille poolest on ta ka rohkem tuntud, kuhu lisati koormusjaoturi funktsionaalsus, et ühe tarkvaraga oleks võimalik luua veebiserver pakkuda sellele koormuse tasakaalustamist. F5 Nginx tarkvara on lihtne paigaldada ja seadistada kuid selle jõudlus kõrge koormuse korral jääb märgatavalt alla HAProxyle.

HAProxy puhul on tegemist kõige tuntuma koormusjaoturiga, mis on tasuta ning mis on lähtekoodtarkvara. HAProxy on olnud turul pikki aastaid ning sellega kujunenud koormusjaoturite standardiks, mis tegeleb ainult koormuse tasakaalustamisega ning selle tulemusel on tekkinud palju erinevaid koormuse tasakaalustamise funktsionaalsusi. Tulenevalt paljudest erinevatest funktsioonidest ja seadistustest võib esialgu koormusjaoturi seadistamine tunduda keerulisem kui teistel. Kuid pärast koormusjaoturi tööle saamist ja seadistamist näitab HAProxy head jõudlust tulles toime ka kõrge koormusega.

Autor hinnangul kõige sobivamaks koormusjaoturiks töö teostamise on HAProxy, kuna see pakub head jõudlust ning rohket funktsionaalsust. Hea jõudlus on eriti oluline, kuna tegemist on ärikriitilise rakendusega, kus periooditi tõuseb koormus väga kõrgeks, mis hetkel on eriti oluline, et päringud rakendusse kohale jõuaksid.

8 Rakenduse üleviimine konteinertehnoloogiale

Rakenduse juurutamisel konteinerisse on vaja teha muudatusi nii rakenduse konfiguratsiooni failides, mis asuvad koodi hoidlas kui ka rakenduse juurutamise töövoos. Rakenduse sujuvaks juurutamiseks luuakse eraldi konteiner koormusjaoturile, mis suunab päringuid nii konteineris töötavale rakendusele kui ka virtuaalmasinas töötavale rakendusele. Koormusjaoturiga saab suunata rohkem päringuid konteinerisse, et veenduda rakenduse toimivuses ja seejärel rakendus sulgeda virtuaalmasinas.

8.1 Konteinerisse kohandamine

Rakenduse konteinerisse kohandamiseks on vajalik rakenduse valmis ehitatud fail sobitada konteinerisse paigaldamiseks. Kui virtuaalmasina puhul on operatsioonisüsteem juba olemas, kuhu rakenduse paigaldatakse, siis konteineri puhul on vajalik eelnevalt valida baas tõmmis, kus on rakenduse toimimiseks vajalik funktsionaalsus olemas. Antud rakenduse puhul valiti tõmmiseks Eclipse Temurin poolt loodud tõmmis, mis on ametlik OpenJDK tõmmise edasiarendus. Tõmmise versiooni valimisile võeti arvesse virtuaalmasinas olev OpenJDK suurversioon 11 ning versioon, mis on kettapinna mahutavuselt võimalikult väike ja mida siiani uuendatakse. Samuti ei võetud arvesse suurversiooni alamversioone, kuna need võivad sisaldada rohkem vigu ja turvanõrkusi, mida ei uuendata. Nendel tingimustel sai valitud tõmmis versiooniga *eclipse-temurin:11-jre-alpine*. Alpine puhul on tegemist Linuxi distributsiooniga, mis on turvalisusele orienteeritud, kergekaaluline ja oma mahult kõige väiksem distributsioon, mida Dockeri tõmmistes kasutatakse [40].

Peale tõmmise välja valimist on vajalik sellega koos luua enda rakenduse spetsiifiline tõmmis. Selle loomist tuleb alustada Dockeri poolt kindlaks määratud failist nimega Dockerfile, kuhu tuleb defineerida, kuidas luuakse Dockeri tõmmis ning kuidas rakendus tööle läheb, kui Dockeri tõmmis käivitatakse konteineris. Failis tuleb kirjeldada eelnevas peatükis mainitud tõmmis, rakenduse nimi, rakenduse kokku pakitud Java arhiivfaili asukoht ja selle nimi, pordi number, läbi mille on võimalik rakendusele ligi saada ja

viimaseks kuidas rakendus konteineris käivitada. Lisa 2 esitab rakenduse spetsiifilist Dockerfile'i.

Lisas 2 esimesel real on kirjeldatud, milline baas tõmmis võetakse kasutusele rakenduse tõmmise loomisel, see määrab algväärtuse uue ehitusetapi loomiseks. Teisel ja kolmandal real olevad ARG väärtused, MAX_COUNT ja PROJECT_NAME defineerivad muutujaid, mida saab tõmmise ehitamisel kaasa anda (Lisa 3). MAX_COUNT on väärtus, mis tähendab, kui kauga hoitakse tõmmiseid registris. PROJECT_NAME on rakenduse nimi, mida kasutatakse ka rakenduse JAR faili nimena. Lisa 2 neljandal real määratakse LABEL, mis lisab tõmmisele metaandmed, mille puhul on tegemist tõmmise säilitamise andmetega. Järgnevatel ridadel viis, kuus, seitse ja kaheksa määratakse keskkonnamuutujad. Nendeks on rakenduse JAR faili nimi, OpenTelemetry jälgitavuse instrumenteerija JAR faili nimi, Java rakenduse käivitamisel ette antavad muutujad ja konteineri ajavöönd. Järgnevalt on määratakse WORKDIR parameetriga töökaust, kus hakatakse järgnevaid tegevusi tegema ja kuhu paigaldatakse rakenduse JAR fail ja OpenTelemetry JAR fail. RUN parameeter käivitab käsud, et uuendada distributsiooni, paigaldada tzdata tööriist, mis võimaldab seadistada ajavööndit, eemaldada uuendamisel tekkinud failid ning luua grupp java ja kasutaja java. Parameetritega ADD lisatakse eelnevalt kirjeldatud JAR failid konteinerisse ja seejärele antakse kasutajale java kõik õigused nendele failidele. USER tähendab, mis on konteineri vaike kasutaja ning mis kasutajaga rakendus käivitatakse. EXPOSE 8082 kirjeldab porti, mida konteiner käivitamisel antud võrgus kuulab. Viimaseks on lisatud parameeter ENTRYPOINT, mis võimaldab seadistada konteiner käivitama failina, mille puhul ei käivitu konteiner kui rakendus tööle ei hakka. ENTRYPOINT juures käivitatakse java JAR rakendus koos parameetritega, millega määratakse java rakenduse parameetrid ja OpenTelemtry tööriista instrumenteerimise parameetrid.

Pärast Dockerfile'i koostamist tuleb selle alusel ehitada rakenduse spetsiifiline tõmmis, mida oleks võimalik konteineris tööle panna. Selleks on ettevõttes juba välja kujunenud skript nimega *build.sh*, mida saab iga rakenduse puhul kasutada. Põhilised osad sellest on tõmmisfaili loomine ning tõmmisfail konteinerite registrisse üles laadimine. Skript pannakse Bamboos tööle peale seda kui rakendus on valmis ehitatud. Lisa 3 esitab rakenduse tõmmise ehitamise skripti.

Lisas 3 esimesel real on defineeritud, kuidas käivitatakse skript ja millise süsteemi kesta (ingl. shell) see skript esindab. Järgnevalt on kui lause, mis kontrollib, kas skripti käivitades on lisatud argument ja kui pole, siis skript lõpetab töö ning kuvab teksti. Muutuja PROJECT_NAME kasutatakse rakenduse tõmmise ehitamisel, millega tekitatakse tõmmiste registrisse vastav kaust kui ka kasutatakse rakenduse JAR faili nimetamisel. PROJECT_TAG on Bamboost saadud rakenduse ehitamise järgunumber koos haru nimega. DOCKER_CMD defineerib käsu nii, et see kirjutaks standardväljundisse tee (ingl. *path*) nime, mida praegune kest kasutab. DOCKER_HOST määrab tõmmiste registri asukoha aadressi, kuhu laetakse üles ehitatud tõmmise fail. Muutuja IMAGE_FILE märgib tekstfaili nime, kuhu kirjutatakse üles laetud tõmmise täispikk nimi, et Bamboo saaks seda kasutada rakenduse juurutamisel, leides vastab tõmmis tõmmiste registrist. CLEANUP_COUNT on number, mida kasutatakse CLEANUP_LABEL jaoks, kirjeldades tõmmise säilitavuse väärtust. Kui-muidu (ingl. *if-else*) lauses kontrollitakse rakenduse hoidla haru nime, millega ehitatakse tõmmist ning vastavalt sellele, kas eemaldatakse haru eest sõna *feature* ja *bugfix* või *release* nime järelt asendatakse kaldkriips sidekriipsuga, viimasel juhul võrdsustatakse muutuja FORMATTED_TAG muutujaga PROJECT_TAG. FORMATTED_TAG määrab ära selle kuidas on tõmmis registris nimetatud. Järgnevas käsus ehitatakse tõmmis kasutades eelnevalt kirjeldatud muutujaid ning viimase käsuga laetakse üles tõmmis registrisse rakenduse nimelisse kausta. Rakenduse tõmmise üles laadimisel uuendatakse ka alati rakenduse tõmmist nimega *latest*, mis on alati rakenduse viimane versioon registris. Viimase käsuna pannakse tekstfaili kirja, tõmmise nimi ja selle asukoht registris.

8.2 Kubernetesesse juurutamine

Kubernetesesse juurutamiseks tuleb kirjeldada YAML formaadis konfiguratsiooni failid. Failidega luuakse erinevad Kubernetese objektid, mis on püsivad üksused Kubernetse süsteemis, mida Kubernetes kasutab klastri seisundi kujundamiseks. Need kirjeldavad, millised konteineriseeritud rakendused töötavad, mis ressursid on neile kätte saadavad ja mis on nende rakenduste käitumise poliitika, näiteks taaskäivitamise poliitika, uuendused ja tõrketaluvus [41]. Kubernetese väikseim juurutatav üksus on kaun, mida saab luua ja hallata. Kaun on ühe või mitme konteineri rühm, millel on ühised salvestus- ja võrguressursid ning spetsifikatsioon selle kohta, kuidas konteinereid hallata [42].

Konteineri käivitamisel on vajalik kaun, kuid seda ei looda otse Kubernetesesse, sest nii ei saa kasutada Kubernetese poolt pakutavaid skaleeritavuse ja tõrkesiirde võimalusi. Selleks, et skaleeritavus ja tõrkesiire saavutada tuleb luua *Deployment* objekt. Rakenduse juurutamisel Kubernetesesse kasutatakse nelja faili, nendeks on vastavalt Kubernetesesse lisamise järjekorras: *secret.yaml*, *configmap.yaml*, *service.yaml* ja *deployment.yaml*. Konfiguratsioonifailis *secret.yaml* on kirjas objekt *Secret* ning saladuste fail nimi. Faili rida *apiVesrion* kirjeldab, mis versiooni Kubernetese API tööriista kasutatakse ja mis on selle eesmärk, et luua objekt. Saladuste objekt lisatakse kubernetesesse esimesena, et enne deployment faili oleksid saladused kättesaadavad, millele rakendus peab ligi saama. Saladused ise selles failis kirjeldatud ei ole kuid asuvad krüpteeritult teises failis. Saladused lisatakse alles juurutamise protsessi jooksul ning lisatakse *secret.yaml* faili, mis seejärele Kubernetesesse lisatakse. Joonis 7 esitab objekti *Secret* konfiguratsioonifaili sisu.

```
kind: Secret
apiVersion: v1
metadata:
  name: credentials
type: Opaque
data:
  application.yml: {APPLICATION_YML}
```

Joonis 5. Objekti *Secret* konfiguratsioonifaili sisu.

Teise failina lisatakse *configmap.yaml* fail, milles on kirjeldatud objekt *ConfigMap*, mis on objekt, mida kasutatakse mittekonfidentsiaalsete andmete salvestamiseks võtme ja väärtuse paaridena [43]. Kaunad võivad väärtusi tarbida keskkonnamuutujatena, käsurea argumentidena või konfiguratsioonifailidena. Fail peab olema samuti enne *deployment.yaml* faili lisatud Kubernetesesse, et *Deployment* lisamisel oleksid keskkonnamuutujate väärtused olemas. Joonis 8 esitab objekti *ConfigMap* konfiguratsioonifaili sisu. Väli *data* all on defineeritud muutujad, mida Kubernetesesse lisatakse.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: properties
data:
  spring.profiles.active: $SPRING_PROFILES_ACTIVE
  deployment.environment: $DEPLOYMENT_ENVIRONMENT
  ci.name: $APPLICATION_CI
  hid.ci: $APPLICATION_HID_CI
  app.namespace: $APP_NAMESPACE
  otel.endpoint: $OTEL_ENDPOINT
```

Joonis 6. Objekti *ConfigMap* konfiguratsioonifaili sisu.

Kubernetese kaunale ja seal olevale konteinerile ligispääsemiseks väljastpoolt Kubernetese võrku on vaja luua võrgukiht, mis seda võimaldab. Selleks on Kuberneteses loodud *Service* tüüpi objekt, mis pakub dünaamilist podide avastamist ja koormuse tasakaalustamist. Samamoodi nagu eelnevad konfiguratsioonifailid algab see objekti nimetusest ning *apiVersion* väärtusest. Järgneval väljal on kirjeldatud metaadnemd, milleks on objekt *Service* nimi, annotatsioonid F5 koormusjaoturile ja selle DNS serverile ning väli *labels*, mis on rakenduse unikaalne nimi, et tuvastada, mis rakendusega on tegemist. Eelnevalt mainitud annotatsioonid on kõikides ettevõtte Kubernetese rakendustel kohustuslikud kui rakendus päeva Kubernetesest väljapoole avatud olema, nende olemasolul peab olema väli *type* väärtusega *LoadBalancer*. See tähendab, et teisi väärtusi *type* väljal kasutada ei saa. Väli ports kirjeldab, mis tüübiga on tegemist ja millist protokolli antud port kasutab. Välja all olev *port* kirjeldab konteineris avatud porti ja väli *targetPort* kirjeldab porti, mis on avatud Kubernetese võrgus ning mille kaudu rakendusele Kubernetese võrgust väljastpoolt konteinerisse ligi pääseb. Joonis 9 esitab objekti *Service* konfiguratsioonifaili sisu.

```
kind: Service
apiVersion: v1
metadata:
  name: "$PROJECT_NAME"
  annotations:
    f5controller.kubernetes/provision: "true"
    f5dnscontroller.kubernetes/provision: "true"
  labels:
    app_ci_hid: "$APPLICATION_HID_CI"
spec:
  selector:
    app: "$PROJECT_NAME"
  type: LoadBalancer
  ports:
    - name: http
      port: 8082
      protocol: TCP
      targetPort: 8082
```

Joonis 7. Objekti *Service* konfiguratsioonifaili sisu.

Lisas 2 on välja toodud faili *deployment.yaml* sisu, kus on defineeritakse *Deployment* objekt. Objekti *Deployment* metaandmete nimi peab iga *Deployment* objekti puhul erinev olema. Väli *replicas* kirjeldab, mitu identset kauna Kubernetese klastris luuakse. *Stradegy* väljal kirjeldatakse rakenduse juurutamise strateegia, mille puhul on tegemist veereva uuendusega (ingl. *rolling update*), mille puhul juurutades luuakse uued kaunad ning vanad kaunad jäävad seni tööle kuni uued on tööle hakanud. See strateegia võimaldab katkestuseta juurutamist, kuid seda ainult kubernetese piires. Väljal *template* kirjeldatakse parameetrid, mis on kõikides kauna koopiates samad, antud juhul on selleks PROJECT_NAME, mis on rakenduse lühinimi ja APPLICATION_HID_CI, mis on rakenduse unikaalne ja muutumatu nimi. Välja container all olevad parameetrid kirjeldavad konteineri nime (*name*), tõmmis faili nime (*image*), konteineri kasutatavaid ressusse (*resources*), rakendus ligisaamiseks avatud porti konteineris (*port*) ja sekstiooni (*volumeMounts*), kus asuvad saladused ning keskkonnamuutujaid (*env*), mida rakendus vajab, et tööle minna. Keskkonnamuutujad määravad rakenduse keskkonna tüübi, rakenduse täisnime ja unikaalse nime ning OpenTelemetry instrumenteerija jaoks vajalikud parameetrid. Konteineri töösolekut kontrollib *livenessProbe* väli, mis kontrollib pidevalt konteineri töösolekut alates sellest kui konteiner käivitatakse. Kui konteiner töösolek katkeb, siis sellele tehakse taaskäivitus ning proovitakse uuesti tööle saada. Lisaks töösoleku kontrollile on defineeritud ka valmisoleku kontrolli väli *readinessProbe*, mis kontrollib, kas konteiner on valmis päringuid vastu võtma.

Töösolekut kontrollitakse, kas rakendus vastab päringule *heartbeat* asukohalt, mis vastab kas rakendus on tööle läinud või mitte ning valmisoleku kontrolli puhul päritakse *health* asukohta. Viimaseks on defineeritud väli *volume,* kus on kirjeldatud sektsiooni nimi ja saladuste objekti nimi.

Eelnevalt välja toodud konfiguratsioonifailid lisatakse Kubernetese keskkonda kasutades bash skripti ning *kubectl* käsurea tööriista. Kubectl tööriist on loodud Kubernetese poolt, mis võimaldab suhelda Kubernetese klastriga läbi API liidese, et hallata ressursse [44]. Skript valib vastavalt keskkonna tüübile krüpteeritud saladuste faili, dekrüpteerib sellelisab ning lisab saladuste objekti konfiguratsioonifaili. Samamoodi valib skript *configMap* muutujate faili ning lisab väärtused objekti konfiguratsioonifaili. Seejärel lisatakse konfiguratsioonifailid *kubectl* käsuga Kubernetesesse.

8.3 Koormusjaoturi juurutamine ja seadistamine

Koormusjaoturite analüüsimisel selgus, et kõige paremini sobib antud töös kasutada koormusjaoturit HAProxy. Kuna antud koormusjaoturil on ametlikult toetatud Dockeri tõmmis, siis sai koormusjaotur sarnaselt rakendusele konteineriseeritud ja juurutatud Kubernetesesse. Nagu ka rakenduse puhul tuleb esmalt luua Dockerfile ning see valmis ehitada ja dockeri registrisse üles laadida. Joonis 8 esitab koormusjaoturi HAProxy koostatud Dockerfile'i.

```
FROM haproxy:lts-alpine
ARG MAX_COUNT
LABEL artifactory.retention.maxCount=${MAX_COUNT}
ENV TZ="Europe/Tallinn"
USER root
RUN apk update \
    && apk upgrade \
    && apk add tzdata \
    && apk add curl \
    && apk add socat \
    && rm -rf /var/cache/apk/*
EXPOSE 8082
```

Joonis 8. Koostatud koormusjaoturi HAProxy Dockerfile.,

Joonis 8 esimesel on kirjeldatud HAProxy baas tõmmist versiooniga *lts-alpine*, kus sõna LTS ehk *long-term support* tähendab kestustuge, mis taotleb kasutamise stabiilsust. Teiseks sõnaks on Alpine ehk tegemist on Alpine distributsiooni baasil oleva tõmmisega,

mis on väikese mahuga ja kergekaaluline. Järgnevalt on failis kirjeldatud samad read, mis rakenduse tõmmises, et koormusjaoturile kehtiksid rakendusele sarnased tõmmise säilitamise reeglid. Samuti on failis kirjeldatu ajavöönd ning paigaldatakse konteinerisse eraldi *tzdata*, *curl* ja *socat* tööriistad. Tööriist *tzdata* võimaldab kasutada ajavööndit konteineris, tööriista *curl* on vajalik paigaldada, et tagada rikkeotsingu võimalus konteineris, kasutades URL-süntaksit ning *socat* tööriist võimaldab muuta HAProxy seadistusi otse konteinerist, ilma konteineri juurutamise protsessi läbi tegemata ning taaskäivituseta kuid tegemist on ajutiste seadistuse muudatustega ehk muudatused salvestatakse muutmälusse, mis taaskäivitamise puhul kaovad. Samuti on kirjeldatud rakendusega sama port, mida ka koormusjaotur konteineri käivitamisel kuulab. Erinevalt rakendusest käivitatakse koormusjaotur kasutajaga *root*, põhjusel paigaldada eelnevalt mainitud tööriistad kuna tõmmise vaikekasutajal polnud selleks õigusi. Koostatud failist Dockeri tõmmise loomisel kasutatakse sama skripti, mida kasutati ka rakenduse loomisel, mis on toodud välja Lisas 3.

Koormusjaoturi juurutamisel Kubernetesesse on samuti võimalik kasutada rakendusele sarnaseid faile, eemaldades rakenduse spetsiifilise seadistuse ning lisades koormusjaoturi juurutamiseks vajaliku seadistuse. Välja jäeti täielikult *Secret* objekt kuna koormusjaoturi seadistused puuduvad saladused. Joonis 9 esitab Kubernetese objekti *ConfigMap* koormusjaoturi konfiguratsioonifaili sisu.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: properties
data:
  deployment.environment: $DEPLOYMENT_ENVIRONMENT
  haproxy.config: |
```

Joonis 9. Objekti *ConfigMap* koormusjaoturi konfiguratsioonifaili sisu.

Joonisel esitatud objekti ConfigMap seadistuses alles väli *deployment.environment* mille väärtuseks on vastav keskkond, kas arendus-, test- või toodangukeskkond. Järgnevalt on defineeritud *haproxy.config* väli, mille väärtuseks on antud juhul püstkriips kuid, millele järgneb HAProxy seadistuse fail. Seadistuse faili sisu lisatakse objektile Kubernetesesse juurutamise jooksul, kasutades rakendusele sarnast skripti Kubernetesesse juurutamisel, erinevusena eemaldati saladuste lisamine ning lisati käsk *'cat haproxy.config >>*

configmap.yaml', kus lisatakse koormusjaoturi konfiguratsioonifaili *haproxy.config* sisu *configmap.yaml* faili.

Kubernetese objekti *Service* konfiguratsioonifail on koormusjaoturi puhul sama, mis rakenduse puhul ning kirjeldatud joonisel 7. Objekti Deployment konfiguratsioonifail on välja toodud lisas 5, mis erineb rakenduse konfiguratsioonifailist mõningate väljade poolest. Esimene erinevus on väli *volumeMounts*, mis rakenduse puhul kirjeldas saladuste asukohta konteineris, ning selle sektsiooni nime, kuid koormusjaoturi puhul kirjeldab väli koormusjaoturi konfiguratsioonifaili asukohta, kust HAProxy saab käivitamisel oma seadistuse. Samuti on koormujaoturi failis ainult üks keskkonnamuutuja, mis kirjeldab keskkonda, kas tegemist on arendus-, test- või toodangukeskkond. Erinevalt rakendusest puudub koormusjaoturil valmisoleku kontroll, mis rakenduse puhul on vajalik kuna peale rakenduse töösolekut käivituvad taustprotsessid enne mida ei saa päringuid rakendusele saata. Koormusjaoturi puhul kontrollib Kubernetes töösolekut healthz asukoha kaudu, mis on seadistatud HAProxy konfiguratsioonifailis. Üks suuremaid erinevusi on väli *volume*, kus on kirjas välja *volumeMounts* nimi, et Kubernetes teaks asukohta, kus HAProxy konfiguratsioonifail asuma hakkab. Välja *configMap* all on kirjeldatud Objekti *ConfigMap* metaandmete nimi ning sellele järgneb *configMap* failis oleva *data* välja all oleva väärtuse nimetus ehk *haproxy.config*. Lisas 5 viimases real on kirjeldatud HAProxy konfiguratsioonifaili nimi *haproxy.cfg*.

HAProxy konfiguratsioonifailis peab olema neli põhilist jaotist, milleks on: *global*, *defaults*, *frontend* ja *backend*. Need neli jaotist määravad, kuidas server tervikuna toimib, millised on vaikimis seaded ning kuidas päringuid vastu võetakse ja suunatakse tagarakendusele [45]. Joonis 10 esitab koormusjaoturi seadistust HAProxy konfiguratsiooni failis.

```
    global
      stats socket /var/run/api.sock user haproxy group haproxy mode 660
level admin expose-fd listeners
      log stdout format raw local0 info

    defaults
      mode http
      timeout connect 10s
      timeout client 30s
      timeout server 30s
      log global

    frontend api
      bind :8082
      monitor-uri /healthz
      default_backend rakendusserverid

    backend rakendusserverid
      balance roundrobin
      option httpchk
      http-check send meth GET uri /health
      http-check expect status 200
      server k8s1 172.168.1.101:8082 check weight 25
      server k8s2 172.168.1.102:8082 check weight 25
      server vm1 192.168.1.103:8082 check weight 100
      server vm2 192.168.1.104:8082 check weight 100
```

Joonis 10. Koormujaoturi seadistus HAProxy konfiguratsioonifailis.

HAProxy konfiguratsioonifaili ülaosas on globaalne osa, mida tähistab sõna *global* oma real. Seadistused *global* all määravad kogu protsessi hõlmava turvalisuse ja jõudluse häälestused, mis mõjutavad HAProxy't madalal tasemel. Joonisel 10 jaotise *global* all esimene rida *stats socket* võimaldab kasutada HAProxy *Runtime API* funktsionaalsust, mille abil saab dünaamiliselt keelata servereid, töösoleku kontrolle ja muuta serverite koormuse tasakaalustamise kaale. Järgneval real on *log* seadistus, mis tagab, et käivitamise ajal väljastatud hoiatused ja töö ajal tekkivad probleemid logitakse Kubernetese logi väljundisse tekst formaadis. Logide tõsiduseaste on pandud *info*, mis puhul jõuavad logi väljundisse päringute info, millega saab kontrollida, et koormusjaoturisse tulevad päringud õnnestuvad või ebaõnnestuvad.

Konfiguratsioonifaili sisu kasvades, aitab *defaults* jaotise kasutamine vähendada dubleerimist. Selle seaded kehtivad kõigile selle järel olevatele esi- ja tagarakenduste osadele. Jaotises *defaults* seadistus *mode http* määrab, kas HAProxy on võimeline kontrollima sissetulevate päringute HTTP sõnumeid. Seadistus *timeout connect* määrab

aja, mille jooksul koormusjaotur ootab, kas ühenduse loomist tagarakenduse serveriga. *Timeout client* seadistus mõõdab tegevusetust perioodidel kui klient saadab TCP segmente. Seadistus *timeout server* mõõdab mitteaktiivsust ajal kui oodatakse edasi suunatud päringu vastust. Timeout seadistuste aeg on kirjeldatud sekundites, mida tähendab ühik „s“ numbrite lõpus. Aja määramisel on võetud arvesse pikima päringu kestus. Jaotise viimasel real on seadistus *log globaal*, mille määramisel kasutab iga *frontend* jaotis logimise seadistust, mis on seatud jaotises *globaal*.

Seadistades HAProxy koormusjaoturiks tagarakenduste ette, määratleb *frontend* jaotis IP aadressid ja pordid, millega kliendid saavad ühendust võtta. Antud jaotise esimene seadistus *bind* määrab kuulaja antud IP aadressile ja pordile, milleks on port 8082, mis on rakendusel avatud. Kuna määartud ei ole kindlat IP aadressi, siis seotakse kõik IP aadressid serveris. Seadistus monitor-uri on koormusjaoturi töösoleku kontrolliks loodud laiend, mida Kubernetes kasutab, et kontrollida koormusjaoturi töösolekut. Viimaseks seadistuseks on *default_backend*, mis kasutab jaotise *backend* nime, et teada, kuhu päringuid saadetakse. Antud töös on jaotise *backend* nimeks „rakendusserverid“.

Jaotis *backend* määratleb serverite rühma, mille koormust tasakaalustatakse ja mis määratakse taotluste töötlemiseks. Igale *backend* jaotisele lisatakse vabalt valitud silt, milleks on antud töös„rakendusserverid“. Jaotises on esmalt määratud *balance roundrobin*, mis kirjeldab, millist koormuse tasakaalustamise algoritmi kasutatakse, milleks on *round robin* algoritm. Järgmiseks on *option httpchk*, mis paneb koormusjaoturi saatma HTTP töösoleku kontrolle rakendusserveritele. Vaikimisi teeb HAProxy töösoleku päringuid juur asukohta poole, kuid lisades *http-check send meth GET uri /health* rea, saadetakse töösoleku kontrolle HTTP GET meetodiga asukoha health poole. Sellest järgnev rida *http-check expect status 200* määrab vastuse mida töösoleku päringu korral oodatakse, antud juhul oodatakse, et vastuseks tuleks staatus 200, mis tähendab õnnestunud päringut. Viimasel neljal real on määratud seadistus *server*, mille esimene argument on nimi, mis antud juhul tähistavad Kuberneteses olevat rakendust (k8s1 ja k8s2), kasutades Kubernetese ametlikku lühendit K8s klastris 1 ja 2 ning rakendust kahes virtuaalmasinas 1 ja 2 (vm1 ja vm2). Nime argumendile järgneb rakendusserveri IP aadress ja port, millega saab rakendusele ligi. Argument *check* määrab, kas antud serverile tehakse töösoleku kontrolli või mitte. Viimasena on kirjeldatud argument *weight*, mida kasutatakse serveri kaalu kohandamiseks võrreldes teiste serverite suhtes. Kõik serverid saavad koormuse, mis on proportsionaalne nende kaaluga võrreldes kõigi kaalude

summaga ehk mida suurem on kaal seda suurem on koormus. Kaalude minimaalne väärtus on 0 ja maksimaalne 256. Väärtuse 0 puhul ei osale server koormuse tasakaalustamises, kuid võtab vastu püsivaid ühendusi. Antud seadistuse puhul on virtuaalmasinas töötavad rakendused 75 võrra suurema kaaluga kui Kubernetese konteineris töötavad rakendused. See omakorda tähendab, et virtuaalmasinas töötavad rakendused on 75% suurema koormusega kui Kubernetese konteineris töötavad rakendused. Sellisel kujul kaalude jaotamine võimaldab algoritmi *weighted round robin* kasutamist kuna kaalud algoritm arvestab serverite kaalusid ega jaga koormust võrdelt kõigi rakenduste vahel.

Päringute suunamiseks koormusjaoturile on vajalik muuta ka varasemalt kasutuses olev DNS kirjet. Selleks muudeti DNS A kirje, kus DNS nimi oli seotud avaliku IP aadressiga, DNS CNAME kirjeks, kasutades sama DNS nime kuid mis on seotud Kubernetese F5 DNS nimega. F5 DNS nimi lisatakse igale *Service* objektile, mis Kuberneteses luuakse. DNS kirje muutmisel jõuavavad päringud Kuberneteses töötavale koormusjaoturile, mis omakorda suunab need edasi vastavalt kaaludele, kas Kubernetese konteineris töötavale rakendusele või virtuaalmasinas töötavale rakendusele, kes päringud vastu võtavad ning vastuse tagasi saadavad.

8.4 Pidevintegratsioon ja pidevvalmiduse töövoog

Rakenduse pidevintegratsioon ja pidevvalmidus töövood on antud rakenduse puhul automatiseeritud Bamboo töövahendiga. Koodi valmis ehitamist alustatakse kui koodihoidlasse on kehtestatud koodi muudatus, mis käivitab automaatselt koodi valmis ehitamise töövoo. Bamboos töövood jagunevad *stage, job* ja *task* nimelisteks, kus *stage* alla saab kirjeldada mitu *job* protsessi ning *job* protsessi alla saab kirjeldada mitu *task* tegevust, millest viimases kirjeldatud *task*, mis käivitab koodi ehitamise skripti. Ehitades rakendus paigaldamaks virtuaalmasinasse on tekitatud üks *stage* protsess, koos ühe *job* protsessiga ning kolme *task* tegevusega. Tegevustena on kirjeldatud rakenduse ning koormusjaoturi koodihoidlate alla laadimine, koodi ehitamine JAR failiks ning ühik testide tegemine.

Võimaldades rakendust kui ka koormusjaoturit konteineriseerida ja juurutada Kubernetesesse tuleb alustada tõmmise loomisest Bamboos. Seda saavutades tekitati juurde üks *stage*, koos kahe *job* protsessiga, milles mõlemas on üks loodud üks *task*. Kuna

tõmmiste ehitamisel, koormusjaotur ega rakendus ei sõltu üksteisest, millest tulenevalt on need Bamboos võimalik lisada sama *stage* alla, mis juhul saavad need protsessid paralleelselt töötada ning seeläbi kiiremini kõik ehitamise protsessid lõpetada.

Juurutamise protsess saab Bamboos käivitada automaatselt kui ehitamise protsessid on lõpetanud või käsitsi käivitades, kui tekib vajadus uus muudatus vastavas arendus-, test- või toodangukeskkonnas uuendada. Konteinerisse juurutamisel alustati koormusjaoturi juurutamist, millele järgnes automaatselt rakenduse juurutamine. Koormusjaotur paigaldamine enne rakendust on vajalik, et koormuse tasakaalustamine töötaks kui rakendust uuendatakse ühes ja siis teises Kubernetese klastris.

8.5 Rakenduse katkestuseta üleviimine

Koormusjaoturiga saavutatud päringute suunamine võimaldab testida konteineriseeritud rakenduse tööd ning stabiilsust, kasutades selleks ettevõttes kasutusel olevat uut jälgitavuse töövahendit. Jälgitavuse töövahendi abiga sai tuvastatud, kas rakendus saab vigu, mis võiksid takistada päringute suunamise ainult konteineriseeritud rakendustele.

Lisas 6 esitatud skeemil tuleb päring sisevõrgus töötavast proksist kasutades seal seadistatud DNS nime, mis on seotud Kuberneteses töötava HAProxy DNS nimega ning suunataks HAProxy DNS nime poole edasi. Järgnevalt suunatakse päring ühele kahest Kuberneteses töötavale koormusjaoturi poole, kasutades *Round Robin* algoritmi. Koormusjaoturis suunatakse päring edasi ühele neljast rakendusserverile, kasutades *Weighted Round Robin* algoritmi, mis omakorda suunab selle serveris rakendusele. Kubernetesesse suunatud päring ei liigu otse rakenduse konteinerisse vaid Kuberneteses *Service* (SVC) objektis seadistatud koormusjaoturile, kasutades *Round Robin* algoritmi, mis suunab päringu Kubernetese kaunas töötavasse konteinerisse ning seejärel jõuab päring rakendusele.

Pärast veendumakse, et Kuberneteses töötaval rakendusel vigu ei teki muudeti päringute kaalu suunates kõik päringud ainult Kuberneteses töötavale rakendusele. Selleks sai kasutatud HAProxy poolt pakutavat Runtime API funktsionaalsust, mis võimaldab HAProxy seadistust muuta otse konteinerist salvestades muudatused muutmälusse. HAProxy Runtime API kasutamine võimaldab päringute kiiret tagasi ümber muutmist kui peaks tekkima teadmata vigu ning samuti võimaldab antud funktsionaalsus näha

HAProxy päringute statistikat kui ka rakenduste staatust kuna HAProxy kontrollib rakenduste töösolekut.

HAproxy Runtime API funktsionaalse kasutamiseks sai konteinerisse paigaldatud tööriist *socat* ning kuna HAPorxy konfiguratsioonifailis sai seadistatud *socket* väärtus, siis koormusjaoturi seadistamise muutmine ei ole vajalik kui Runtime API kasutamiseks on vajalik esmalt *socat* tööriistale ette anda HAProxy socket asukoht, käsuga *'socat readline /var/run/api.sock'*, et antud tööriist suudaks HAProxy seadistust muuta. Kasutades *socat* tööriista, et muuta rakendustele saadetavate päringute kaalu, muudeti koormusjaoturi seadistus. Selleks, et päringud liiguksid ainult konteineris töötavale rakendusele käivitati käsk *'echo "set server rakendusserverid/vm1 weight 0" | socat stdio /var/run/hapee-lb.sock'*, mis seadistas ühe virtuaalmasinal töötava rakendusserveri kaaluks 0. Antud käsku tuli käivitada kahes Kubernetese klastris töötavas koormusjaoturi konteineris ning kahe virtuaalmasina kohta, et ajutiselt suunata kõik päringud Kuberneteses töötavale rakendusele. Peale ajutiselt seadistuse muutmist tuli vastav seadistus kinnitada, et muudatus konteineri taaskäivitamisel vana seadistuse peale tagasi ei läheks. Selleks tuli sama muudatus kirjeldada HAProxy konfiguratsioonifailis, mis asus koodihoidlas ning see muudatus Kuberneteses asuvasse koormusjaoturisse juurutada. Lisa 7 kirjeldab päringute liikumist pärast seadistuse muutmist.

9 Tulemused

Rakenduse konteineriseerimisel ja juurutamisel Kubernetesesse saavutati katkestuseta lisades rakendusele lisaks ka koormusjaotur konteinerisse, mis võimaldas suunata sisse tulevad päringud nii virtuaalmasinas töötavatele rakendustele kui ka Kubernetese konteineris töötavatele rakendustele. Järgnevalt muudeti seadistust suunates päringud ainult konteineris töötavale rakendusele, mille vastav skeem on välja toodud lisas 7. Selliselt koormusjaoturit kasutades saavutati rakenduse sujuv üleminek antud ettevõttes virtuaalmasinast konteinerisse ning lahendati rakenduse jälgitavuse probleem.

Hetkel, kus rakendused töötasid konteineris kui ka virtuaalmasinas, oli kokku 4 rakendust, mis võtsid päringuid vastu ja töötlesid neid. Selle tulemusel suurenes andmebaaside ühenduste arv kahekordselt, mis lisas omakorda koormust andmebaasile kuid andmebaasiga probleeme ei tekkinud. Andmebaasi jõudlus kinnitab, et Kubernetese kaunade arvu tõstes võib kindel olla andmebaasi jõudluses, mis on üks põhikomponente rakenduse toimimisel.

Kubernetesesse rakenduse kui ka HAProxy koormusjaoturi juurutamisel ning Kubernetese koormusjaoturite kasutamisel saavutati staatiliste koormuse tasakaalustamise algoritme kasutamisel dünaamiline koormuse tasakaalustamine, kus rakenduse üles ja alla skaleerimisel pole vajalik eraldi HAProxy konfiguratsioonifaili seadistust muuta, et lisatud rakendustele päringud jõuaksid kuna HAProxy päringud on suunatud rakenduse Kubernetese *Service* objektis kirjeldatud koormusjaoturi poole.

Rakenduse ja koormusjaoturi juurutamisel kasutatud pidevintegratsioon ja pidevvalmidus tööriistade kasutamine võimaldas säilitada arendusprotsessi toimimist ehk rakenduse ehitamisel ja juurutamisel lisatud etapid ja ülesanded ei takistanud arendust ning arendusprotsessi vaatest probleeme ei esinenud. Samuti ei pikenenud märgatavalt ehitamise ja juurutamisele kuluv aeg, kuid CI/CD töövoog lihtsustas rakenduse ja koormusjaoturi juurutamist Kubernetesesse.

10 Järeldused

Antud tööst võib järeldada, et koormusjaoturi kasutamine on vajalik, kui riskide maandamiseks soovitakse rakendust hoida töös nii konteineris kui ka virtuaalmasinas samaaegselt, et tagada rakenduse katkestuseta üleminek konteinerisse. Kuna tegemist oli võrdlemisi keeruka rakendusega, mille kõiki seoseid ei ole täpselt dokumenteeritud, sai otsustatud rakenduse üleviimisel kasutada koormusjaoturit, mis võimaldab päringute suunamist nii ühes kui teises keskkonnas töötavale rakendusele. Samuti sai lahendatud töös tõstatatud probleem, mis nõudis jälgitavuse lahenduse välja vahetamist, et tagada rakenduse jälgitavus ka siis, kui eelmine töövahend enam ei tööta.

Sarnaselt antud rakendusele on võimalik kasutada koormusjaoturit ka teiste rakenduste puhul, mida on vaja konteineriseerida ning mis siiani virtuaalmasinas töötavad. Sellega on võimalik virtuaalmasinad sulgeda ja vabastada nende alla kuuluva ressursi kuna virtuaalmasinad võtavad rohkem ressurssi kui konteineris töötav rakendus Samuti väheneb haldamist vajavate virtuaalmasinate arv, millel peab tagama ajakohasuse ja turvalisuse. Selliselt tagatakse rakenduste ühtluse ja turvalisus läbi arendusprotsessi.

Rakenduse Kubernetesesse konteineriseerimine võimaldab rakenduse juurutamisel kasutusele võtta ettevõttes oleva Kanaari juurutamise meetodi, mis abistaks arendajate koodi muudatuste testimist ja koodi uuendamise puhul võimalike vigade leidmist kui plaanitakse rakendust toodangukeskkonda juurutada. Erinevalt Kanaari juurutamise meetodist võimaldaks ka ainult arendusprotsessi töövoo muutmine saavutada rakenduse parema juurutamise, automatiseerides kogu töövoog alates rakenduse ehitamisest kuni Kubernetesesse juurutamiseni.

11 Kokkuvõte

Käesoleva töö eesmärk oli virtuaalmasinal töötav rakendus konteineriseerimine kasutades selleks ettevõttes kasutusel olevat konteineriseerimise lahendust, konteinerite orkestreerijat ning pidevintegratsioon ja pidevvalmiduse tööriistu. Võrrelda koormusjaotureid, mis sobiks rakenduse sujuvaks üleviimiseks konteinerisse, saavutades sellega rakenduse jälgitavuse toimimise kui varasemalt kasutuses olev jälgitavuse tarkvara suletakse.

Töö lahendatavaks probleemiks oli vahetada rakenduse jälgitavuse töövahend, võttes kasutusele ettevõttes etteantud uuem lahendus, seoses vana lahenduse sulgemisega. Töö käigus muudeti rakendust vastavalt, et rakenduse jälgitavus säiliks ning seda oleks võimalik konteineriseerida ja juurutada konteinerite orkestreerija keskkonda ning tehes seda nii, et sama rakendus töötaks konteineris samaaegselt koos virtuaalmasinal töötava rakendusega. Selleks analüüsiti kolme koormusjaoturit, millest valiti välja üks, mis suunas rakendusele tulevaid päringuid vastavalt seadistatud kaaludele.

Väljatöötatud lahendus on loodud konkreetse rakenduse üleviimiseks konteinerisse kuid töös kirjeldatud lahendust on võimalik kasutada ka teiste rakenduste konteineriseerimiseks antud ettevõttes.

Töös püstitatud eesmärk sai täidetud. Selle tulemusena on rakendus konteineriseeritud Kubernetese keskkonnas, kasutades rakenduse juurutamise pideintegratsioon ja pidevvalmiduse tööriistu. Välja valitud koormusjaoturi HAProxy juurutamisel saavutati rakenduse katkestuseta üleviimine virtuaalmasinalt konteinerisse, kasutades päringute suunamiseks HAProxy poolt pakutud funktsionaalsust suunata päringuid vastavalt seadistatud kaaludele. Sellega lahendati töös püsitatud rakenduse jälgitavuse probleem.

Kasutatud kirjandus

[1] „IT ja sidetehnika seletav sõnaraamat“, *e-Teatmik*. http://vallaste.ee/index.asp (vaadatud 29. november 2022).

[2] „Standardipõhine tarkvaratehnika sõnastik“, *STATS*. https://stats.cyber.ee/#showlist (vaadatud 30. november 2022).

[3] „Andmekaitse ja infoturbe leksikon“, *AKIT*. https://akit.cyber.ee/ (vaadatud 30. november 2022).

[4] M. Portnoy, *Virtualization essentials*, Second edition. Indianapolis, Indiana: John Wiley & Sons, 2016.

[5] „What Is Container Technology?“, *SolarWinds*. https://www.solarwinds.com/resources/it-glossary/container (vaadatud 6. november 2022).

[6] „Containerization“, *IBM*, 23. juuni 2021. https://www.ibm.com/cloud/learn/containerization (vaadatud 6. november 2022).

[7] Huawei Technologies Co., Ltd., *Cloud Computing Technology*. Singapore: Springer Nature Singapore, 2023. doi: 10.1007/978-981-19-3026-3.

[8] A. Hohn, *The book of Kubernetes: a complete guide to container orchestration*. San Francisco: No Starch Press, 2023.

[9] „Overview“, *Kubernetes*. https://kubernetes.io/docs/concepts/overview/ (vaadatud 12. märts 2023).

[10] „Orchestration“, *Docker Docs*, 7. november 2022. https://docs.docker.com/get-started/orchestration/ (vaadatud 7. november 2022).

[11] „What is container orchestration?“, *Red Hat*. https://www.redhat.com/en/topics/containers/what-is-container-orchestration (vaadatud 7. november 2022).

[12] „What is Container Orchestration?“, *IBM*, 21. juuni 2021. https://www.ibm.com/cloud/learn/container-orchestration (vaadatud 7. november 2022).

[13] K. Takahashi, „A Study on Portable Load Balancer for Container Clusters,“ Ph.D. dissertatsioon, Informaatika osakond, The Graduate University for Advanced Studies, Hayama, Japan, 2019. [Online]. Kättesaadav: https://www.researchgate.net/publication/337971562_A_Study_on_Portable_Load_Balancer_for_Container_Clusters

[14] J. P. DeMuro, „What is container technology?“, *TechRadar*, 18. detsember 2019. https://www.techradar.com/news/what-is-container-technology (vaadatud 8. november 2022).

[15] „Containers vs Virtual Machines Differences, Pros, & Cons“, *Engine Yard*, 9. oktoober 2020. https://www.engineyard.com/blog/containers-vs-virtual-machines-differences-pros-cons/ (vaadatud 8. november 2022).

[16] S. Psarris, „Containers and Security, Part 2: Benefits and Challenges“, *Reblaze*, 12. august 2021. https://www.reblaze.com/blog/cloud-security/containers-and-security-part-2-benefits-and-challenges/ (vaadatud 8. november 2022).

[17] P. Rombouts, „Container Cons: The complexities associated with Containers“, *SogetiLabs*, 12. november 2019. https://labs.sogeti.com/container-cons-the-complexities-associated-with-containers/ (vaadatud 8. november 2022).
[18] „Container Monitoring: Why, how, and what to look out for“, *Amazon Web Services, Inc.* https://aws.amazon.com/cloudwatch/container-monitoring/ (vaadatud 14. november 2022).
[19] „What is CI/CD?“, *Red Hat*. https://www.redhat.com/en/topics/devops/what-is-ci-cd (vaadatud 14. november 2022).
[20] M. Shahin, M. Ali Babar, ja L. Zhu, „Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices“, *IEEE Access*, kd 5, lk 3909–3943, 2017, doi: 10.1109/ACCESS.2017.2685629.
[21] „What is Continuous Integration“, *Atlassian*. https://www.atlassian.com/continuous-delivery/continuous-integration (vaadatud 18. november 2022).
[22] „What is Continuous Integration?“, *Amazon Web Services*. https://aws.amazon.com/devops/continuous-integration/ (vaadatud 18. november 2022).
[23] „What is Continuous Delivery?“, *Amazon Web Services*. https://aws.amazon.com/devops/continuous-delivery/ (vaadatud 19. november 2022).
[24] „Continuous Delivery“, *Atlassian*. https://www.atlassian.com/continuous-delivery (vaadatud 19. november 2022).
[25] „Continuous integration vs. delivery vs. deployment“, *Atlassian*. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (vaadatud 19. november 2022).
[26] M. Labouardy, *Pipeline as code: continuous delivery with Jenkins, Kubernetes, and Terraform*. Shelter Island, NY: Manning Publications Co, 2021.
[27] C. Majors, L. Fong-Jones, ja G. Miranda, *Observability engineering: achieving production excellence*, First edition. Beijing Boston Farnham Sebastopol Tokyo: O’Reilly, 2022.
[28] „Observability Primer“, *OpenTelemetry*. https://opentelemetry.io/docs/concepts/observability-primer/ (vaadatud 3. märts 2023).
[29] P. Membrey, D. Hows, ja E. Plugge, *Practical Load Balancing: Ride the Performance Tiger*. Berkeley, CA: Apress, 2012.
[30] „What is load balancing? | How load balancers work“, *Cloudflare*. https://www.cloudflare.com/learning/performance/what-is-load-balancing/ (vaadatud 27. märts 2023).
[31] D. A. Shafiq, N. Z. Jhanjhi, ja A. Abdullah, „Load balancing techniques in cloud computing environment: A review“, *J. King Saud Univ. - Comput. Inf. Sci.*, kd 34, nr 7, lk 3910–3933, juuli 2022, doi: 10.1016/j.jksuci.2021.02.007.
[32] „Best Load Balancing Software for Medium-Sized Businesses in 2023“, *G2*. https://www.g2.com/categories/load-balancing (vaadatud 30. märts 2023).
[33] „Free LoadMaster Load Balancer“, *Kemp LoadMaster*. https://freeloadbalancer.com/ (vaadatud 13. aprill 2023).
[34] „HAProxy“, *HAProxy*. https://www.haproxy.org/ (vaadatud 13. aprill 2023).
[35] „Welcome to NGINX Wiki!“, *Nginx*. https://www.nginx.com/resources/wiki/ (vaadatud 13. aprill 2023).

[36] A. Johansson, „HTTP Load Balancing Performance Evaluation of HAProxy, NGINX, Traefik and Envoy with the Round-Robin Algorithm,“ B.S. diplomitöö, Skövde ülikool, Skövde, Sweden, 2022. [Online]. Kättesaadav: http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-21475
[37] „Load Balancing Algorithms, Types and Techniques“, *Kemp LoadMaster*. https://kemptechnologies.com/load-balancer/load-balancing-algorithms-techniques (vaadatud 14. aprill 2023).
[38] D. Skrba, „Fundamentals: Load Balancing and the Right Distribution Algorithm for You“, *HAProxy Technologies*, 2. september 2022. https://www.haproxy.com/blog/fundamentals-load-balancing-and-the-right-distribution-algorithm-for-you/ (vaadatud 14. aprill 2023).
[39] „HTTP Load Balancing“, *Nginx Docs*. https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/ (vaadatud 14. aprill 2023).
[40] „Alpine Linux“, *Alpine Linux*. https://alpinelinux.org/about/ (vaadatud 15. aprill 2023).
[41] „Understanding Kubernetes Objects“, *Kubernetes*. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (vaadatud 16. aprill 2023).
[42] „Pods“, *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/ (vaadatud 16. aprill 2023).
[43] „ConfigMaps“, *Kubernetes*. https://kubernetes.io/docs/concepts/configuration/configmap/ (vaadatud 16. aprill 2023).
[44] „Command line tool (kubectl)“, *Kubernetes*. https://kubernetes.io/docs/reference/kubectl/ (vaadatud 16. aprill 2023).
[45] C. Lavoie, „The Four Essential Sections of an HAProxy Configuration“, *HAProxy Technologies*, 24. oktoober 2018. https://www.haproxy.com/blog/the-four-essential-sections-of-an-haproxy-configuration/ (vaadatud 19. aprill 2023).

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

Mina, Martin Kokk

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Virtuaalmasinal töötava rakenduse katkestuseta konteinerisse juurutamine ühe ettevõtte näitel“, mille juhendaja on Lauri Anton
   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 – Koostatud rakenduse spetsiifiline Dockerfile

```
FROM eclipse-temurin:11-jre-alpine
ARG MAX_COUNT
ARG PROJECT_NAME
LABEL artifactory.retention.maxCount=${MAX_COUNT}
ENV APP_JAR=${PROJECT_NAME}.jar
ENV OTEL_JAR="opentelemetry-javaagent.jar"
ENV JAVA_OPTS="-XX:-OmitStackTraceInFastThrow -XX:MinRAMPercentage=50 -
XX:MaxRAMPercentage=75"
ENV TZ="Europe/Tallinn"
WORKDIR /opt/app
RUN apk update \
  && apk upgrade \
  && apk add tzdata \
  && apk add curl \
  && rm -rf /var/cache/apk/* \
  && addgroup -S -g 8000 java \
  && adduser -S -s /sbin/nologin -H -u 8000 java java
ADD $APP_JAR ./$APP_JAR
ADD otel/$OTEL_JAR ./$OTEL_JAR
RUN chown -R java:java /opt/app
USER java
EXPOSE 8082
ENTRYPOINT exec java $JAVA_OPTS \
  -Duser.language=en \
  -Duser.timezone='Europe/Tallinn' \
  -javaagent:$OTEL_JAR \
  -Dotel.metrics.exporter=$OTEL_METRICS_EXPORTER \
  -Dotel.traces.exporter=$OTEL_TRACES_EXPORTER \
  -Dotel.exporter.otlp.traces.endpoint=$OTEL_EXPORTER_OTLP_TRACES_ENDPOINT \
  -Dotel.exporter.otlp.traces.protocol=$OTEL_EXPORTER_OTLP_TRACES_PROTOCOL \
  -
Dotel.resource.attributes=service.name=$OTEL_SERVICE_NAME,service.namespace=$
OTEL_SERVICE_NAMESPACE,service.instance.id=$OTEL_SERVICE_INSTANCE_ID,deployme
nt.environment=$DEPLOYMENT_ENVIRONMENT \
  -Dotel.log.level=$OTEL_LOG_LEVEL \
  -Dotel.javaagent.debug=$OTEL_AGENT_DEBUG \
  -jar $APP_JAR
```

Lisa 3 – Rakenduse tõmmise ehitamise skript

```
#!/bin/bash

if [ $# -ne 1 ]; then
  echo 'Supply project name as first argument'
  exit 1
fi

PROJECT_NAME="$1"
PROJECT_TAG="${bamboo_planRepository_branchName:?}-${bamboo_buildNumber:?}"
DOCKER_CMD=$(command -v docker)
DOCKER_HOST="docker.registry"
IMAGE_FILE="image.txt"
CLEANUP_COUNT="5"
CLEANUP_LABEL="MAX_COUNT=${CLEANUP_COUNT}"

if [[ ${PROJECT_TAG} == feature/* ]]; then
  FORMATTED_TAG=$(echo ${PROJECT_TAG} | tr '[:upper:]' '[:lower:]' | sed
's#feature/##g')
elif [[ ${PROJECT_TAG} == bugfix/* ]]; then
  FORMATTED_TAG=$(echo ${PROJECT_TAG} | tr '[:upper:]' '[:lower:]' | sed
's#bugfix/##g')
elif [[ ${PROJECT_TAG} == release/* ]]; then
  FORMATTED_TAG=$(echo ${PROJECT_TAG} | tr '/' '-')
else
  FORMATTED_TAG=${PROJECT_TAG}
fi

${DOCKER_CMD} build \
  --force-rm=true \
  -t "$DOCKER_HOST/$PROJECT_NAME:$FORMATTED_TAG" \
  -t "$DOCKER_HOST/$PROJECT_NAME:latest" \
  --build-arg "version=$FORMATTED_TAG" \
  --build-arg "PROJECT_NAME=$PROJECT_NAME" \
  --build-arg "${CLEANUP_LABEL}" ./
${DOCKER_CMD} push "$DOCKER_HOST/$PROJECT_NAME:latest" && ${DOCKER_CMD} push
"$DOCKER_HOST/$PROJECT_NAME:$FORMATTED_TAG"
echo "$DOCKER_HOST/$PROJECT_NAME:$FORMATTED_TAG" > ${IMAGE_FILE}
```

Lisa 4 – Kubernetese objekti *Deployment* konfiguratsiooni fail

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: "$PROJECT_NAME"
  labels:
    app: "$PROJECT_NAME"
    app_ci_hid: "$APPLICATION_HID_CI"
spec:
  replicas: $REPLICA_COUNT
  revisionHistoryLimit: 2
  progressDeadlineSeconds: 180
  selector:
    matchLabels:
      app: "$PROJECT_NAME"
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      name: "$PROJECT_NAME"
      labels:
        app: "$PROJECT_NAME"
        app_ci_hid: "$APPLICATION_HID_CI"
    spec:
      containers:
        - name: "$PROJECT_NAME"
          image: "$APP_IMAGE"
          resources:
            requests:
              cpu: "$RESOURCE_CPU_REQUEST"
              memory: "$RESOURCE_MEMORY_REQUEST"
            limits:
              cpu: "$RESOURCE_CPU_LIMIT"
              memory: "$RESOURCE_MEMORY_LIMIT"
          ports:
            - containerPort: 8082
              protocol: TCP
          volumeMounts:
            - mountPath: /opt/app/config
              name: credentials
          env:
            - name: SPRING_PROFILES_ACTIVE
              valueFrom:
```

```
        configMapKeyRef:
          name: properties
          key: spring.profiles.active
      - name: APPLICATION_CI
        valueFrom:
          configMapKeyRef:
            name: properties
            key: ci.name
      - name: APPLICATION_HID_CI
        valueFrom:
          configMapKeyRef:
            name: properties
            key: hid.ci
      - name: OTEL_METRICS_EXPORTER
        value: none
      - name: OTEL_EXPORTER_OTLP_TRACES_ENDPOINT
        valueFrom:
          configMapKeyRef:
            name: properties
            key: otel.endpoint
      - name: OTEL_SERVICE_NAME
        valueFrom:
          configMapKeyRef:
            name: properties
            key: ci.name
      - name: OTEL_SERVICE_NAMESPACE
        valueFrom:
          configMapKeyRef:
            name: properties
            key: app.namespace
      - name: OTEL_SERVICE_INSTANCE_ID
        valueFrom:
          configMapKeyRef:
            name: properties
            key: hid.ci
      - name: DEPLOYMENT_ENVIRONMENT
        valueFrom:
          configMapKeyRef:
            name: properties
            key: deployment.environment
      - name: OTEL_TRACES_EXPORTER
        value: otlp
      - name: OTEL_EXPORTER_OTLP_TRACES_PROTOCOL
        value: grpc
      - name: OTEL_LOG_LEVEL
        value: info
      - name: OTEL_AGENT_DEBUG
        value: "false"
    livenessProbe:
      httpGet:
        path: "/heartbeat"
```

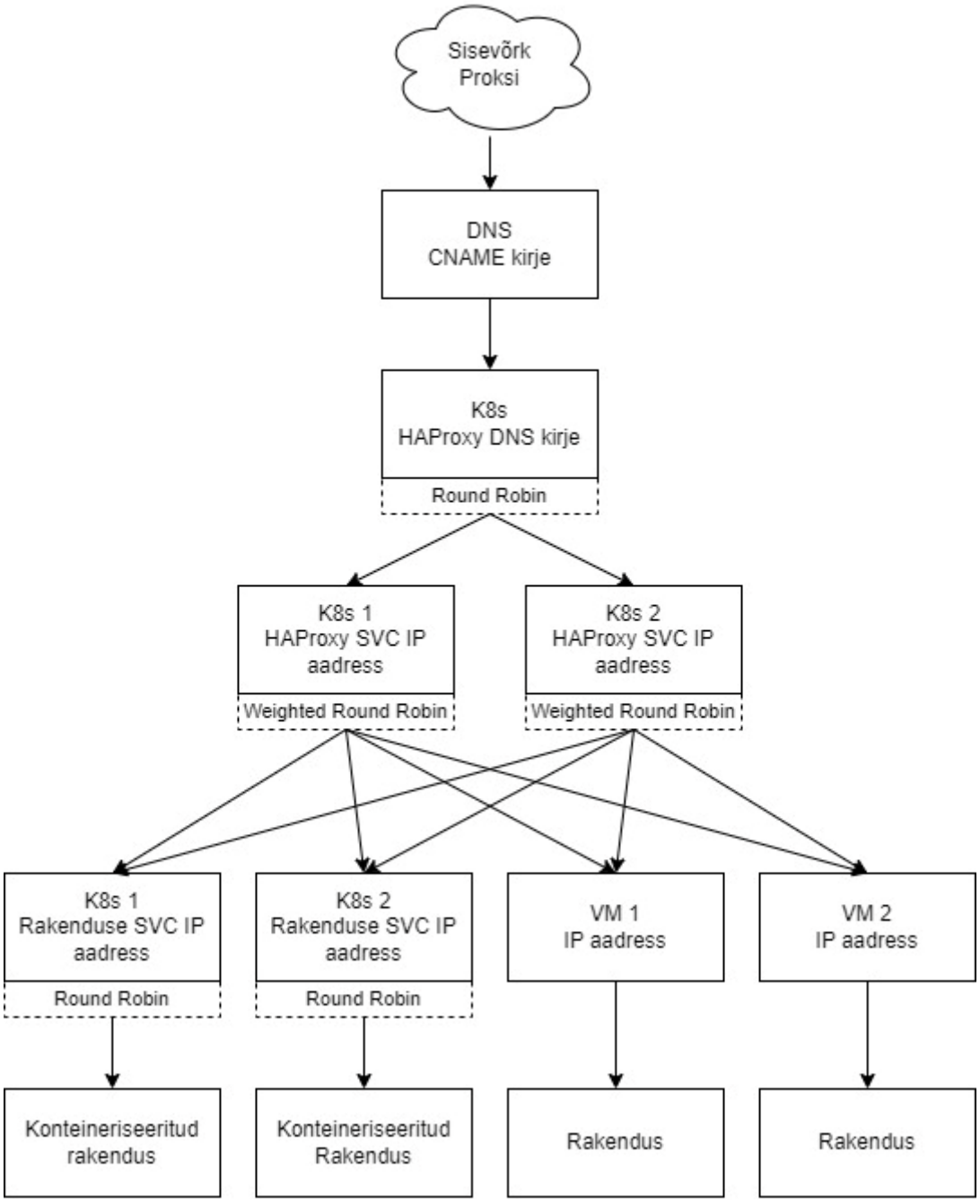
```
        port: 8082
      initialDelaySeconds: 30
      timeoutSeconds: 1
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: "/health"
        port: 8082
      initialDelaySeconds: 60
      timeoutSeconds: 1
      periodSeconds: 10
volumes:
  - name: credentials
    secret:
      secretName: credentials
```

Lisa 5 – Kubernetese objekti *Deployment* koormusjaoturi konfiguratsioonifail

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "$PROJECT_NAME"
  labels:
    app: "$PROJECT_NAME"
    app_ci_hid: "$APPLICATION_HID_CI"
spec:
  replicas: $REPLICA_COUNT
  revisionHistoryLimit: 2
  progressDeadlineSeconds: 180
  selector:
    matchLabels:
      app: "$PROJECT_NAME"
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
  template:
    metadata:
      name: "$PROJECT_NAME"
      labels:
        app: "$PROJECT_NAME"
        app_ci_hid: "$APPLICATION_HID_CI"
    spec:
      containers:
        - name: "$PROJECT_NAME"
          image: "$APP_IMAGE"
          resources:
            requests:
              cpu: "$RESOURCE_CPU_REQUEST"
              memory: "$RESOURCE_MEMORY_REQUEST"
            limits:
              cpu: "$RESOURCE_CPU_LIMIT"
              memory: "$RESOURCE_MEMORY_LIMIT"
          ports:
            - containerPort: 8082
              protocol: TCP
          volumeMounts:
            - mountPath: /usr/local/etc/haproxy/
              name: haproxy
```

```
    env:
      - name: DEPLOYMENT_ENVIRONMENT
        valueFrom:
          configMapKeyRef:
            name: properties
            key: deployment.environment
    livenessProbe:
      httpGet:
        path: "/healthz"
        port: 8082
      initialDelaySeconds: 30
      timeoutSeconds: 10
      periodSeconds: 10
volumes:
  - name: haproxy
    configMap:
      name: properties
      items:
      - key: haproxy.config
        path: haproxy.cfg
```

Lisa 6 – Päringute liikumise skeem pärast rakenduse konteineriseerimist ja koormusjaoturi lisamist



Lisa 7 – Päringute liikumise skeem pärast päringute suunamist konteineris töötavale rakendusele