TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Andre Anijärv

DeepMOOC platvormile tarkvarakonveieri arendamine

Bakalaureusetöö (9 EAP)

Juhendajad:
Tõnis Hendrik Hlebnikov,
Ahti Põder

Tartu 2022

DeepMOOC platvormile tarkvarakonveieri arendamine

Lühikokkuvõte:

DeepMOOC platvorm on loodav keskkond tudengitele ja õppejõududele esitatud programmikoodi automaattestimiseks. Platvormi idee sündis asjaolust, et praegu Tartu Ülikoolis kasutusel olev lahendus - Virtual Programming Lab on piiratud programmeerimiskeelte toega ning ei sisalda mõningaid võimalusi, mis oleks kasulikud programmeerimisaineid läbi viies. DeepMOOC platvormi eesmärk on need piirangud kaotada ning tulevikus saada universaalseks platvormiks, kus läbi viia programmeerimisainete raames automatiseeritavaid tegevusi. Need ei pea piirduma ainult klassikalise ühiktestimisega, vaid võivad endast kujutada näiteks ka koodi staatilist analüüsimist või ajakulu mõõtmist. Selle bakalaureusetöö raames arendatakse DeepMOOC platvormile tarkvaralist konveierit. Konveieri ülesanne on vastu võtta sissetulevad töid ning käivitada neid isoleeritud konteinerkeskkonnas ning on seega üks platvormi põhikomponente. Töös arutletakse ka tehnoloogiliste ja disaini puudutavate valikute üle ning kirjeldatakse lõpptulemuse ülesehitust ning valminud funktsionaalsust. Töö tulemusena valmis kahe konveieri komponendi lähtekood: toru ja tööline. Viimase ülesanne on esitatud koodi jooksutamine isoleeritud Kubernetese *pod*is, kuid toru ise on disainitud olema hõlpsasti laiendatav uut tüüpi töölismoodulitega, mida tehes saab tulevikus platvormile uut funktsionaalsust lisada.

Võtmesõnad: tarkvaraarendus, tarkvaratehnika, Kubernetes, programmeerimisõpe

CERCS: P175 Informaatika, süsteemiteooria

Pipeline development for the DeepMOOC platform

Abstract:

DeepMOOC platform is a new environment in the making for the students and lecturers to carry out automatic testing of the submitted program code. The idea for the platform came from the fact that the current solution used in the University of Tartu – Virtual Programming Lab has limited support for programming languages and is missing some features that would be useful in programming subjects. The DeepMOOC platform aims to eliminate these constraints and eventually become the universal go-to platform for all kinds of automated tasks that can be performed in the context of programming subjects. These do not have to be limited to regular unit-testing but can, for example, include statical code analysis and time measurement. This thesis describes the process of developing the pipeline component for the DeepMOOC platform. The pipeline takes care of receiving the incoming jobs and running them in isolated container environment and is therefore one of the core components of the platform. This paper also discusses the technological and design choices that were made and describes the structure and functionality of final software. The result of this thesis is the source code of the two components of the pipeline: the pipe itself and a worker node that is designed to run submitted code in an isolated Kubernetes pod. The pipe is designed to be modular, so that additional types of worker nodes with new functionalities can be added in the future.

Keywords: software development, software engineering, Kubernetes, programming courses

CERCS: P175 Informatics, systems theory

Sisukord

Sissejuhatus

DeepMOOC on loodav keskkond tudengitele ja õppejõududele programmeerimisalase hariduse omandamiseks ja edasiandmiseks. Platvormi idee on muuta programmeerimisalast õpet huvitavamaks ja kaasahaaravamaks tekitades tudengite lahenduste vahel võistlusmomenti, millel on potentsiaali parandada õppetulemusi [1].

Edetabelite loomine olemasoleva automaatkontrolli lahenduse VPL (Virtual Programming Lab) [2] juurde Tartu Ülikooli programmeerimisainetes "Algoritmid ja andmestruktuurid" ja „Programmeerimine II" on juba praeguseks näidanud tudengite motivatsiooni kasvu programmeerimisalaste kontseptsioonide omandamisel[1]. DeepMOOC platvorm arendab seda ideed edasi, võimaldades õppejõududel püstitada ülesandeid, millele esitatavat koodi testitakse ning analüüsitakse automaatselt. Tulemuste põhjal tekib jooksev edetabel, mille pingerida moodustub esitatud lahenduste korrektsuse, tööaegade, ajaliste keerukuste või muude taoliste väärtuste alusel.

Võrreldes juba olemasoleva ja Tartu Ülikoolis kasutatava lahendusega VPL, püüab DeepMOOC olla oluliselt kasutajasõbralikum, võimaldades automaatkontrolli lihtsamini ja teistele arusaadavamalt üles seada. Lisaks ei piirdu see tavalise ühiktestimisega (*unit-testing*), vaid on VPList oluliselt võimsam, võimaldades automaatkontrolliks üles seada mistahes tegevust. Näiteks on sellega võimalik kontrollida kasvõi tudengi loodud andmebaasi struktuuri või teostada juba mainitud mõõtmisi. Seega saab platvormi valmides automaatkontrolli kasutusele võtta ka ainetes, kus see seni VPLi piirangute tõttu pole võimalik olnud.

Platvormi arendamisega seoses on samaaegselt selle tööga valmimas veel kolm lõputööd, mis puudutavad DeepMOOC platvormi arhitektuuri, eesrakenduse (*frontend*) ja ülejäänud tagarakenduse (*backend*) arendamist. Selle bakalaureusetöö raames arendatakse DeepMOOC platvormi tagarakenduse osana tudengite lahendusi hindavat tarkvaralist konveierit (*pipeline*). Konveieri ülesanne on võtta vastu tudengite esitatud kood, käivitada testid isoleeritud konteinerkeskkonnas ning väljastada tulemus. Tegevustele, mida õppejõud soovib testkeskkonnas teha, ei seata mingeid piiranguid, mis võimaldabki platvormi rakendada ka selliste õppeainete raames, kus praegune lahendus VPL hätta jääb. Töö eesmärk on konveier saada valmis piisavas

[1] Põhineb selle töö juhendajate kogemusel, mis on saadud mainitud õppeaineid läbi viies.

mahus, et selle saaks ühendada tarkvara teiste komponentidega ja et nende valmides saaks platvormi kasutusele võtta.

Töö on struktureeritud kolmeks peatükiks. Teoreetilise ülevaate peatükis võrreldakse DeepMOOC platvormi juba olemasolevate lahendustega, seletatakse ja põhjendatakse konveieris kasutatavaid tehnoloogiaid ja antakse ülevaade konveieri üldisest tööpõhimõttest. Tööprotsessi puudutav peatükk tutvustab samme, mis kulus tarkvaralahenduseni jõudmiseks, arutatakse ka esialgseid ebaõnnestunud ideid ja nendest loobumise põhjuseid. Tulemust käsitlevas peatükis analüüsitakse valminud platvormi komponenti, millised on selle puudujäägid ja võimalikud edasiarendused.

1. Teoreetiline ülevaade

Selles peatükis uuritakse lühidalt DeepMOOC platvormile sarnaste olemasolevate lahenduste ülesehitust ning kas ja mismoodi kasutavad need tudengite programmide jooksutamiseks isoleeritud keskkondi või konteinereid. Seejärel antakse ülevaade käesoleva töö raames arendatavast tagarakenduse konveierist ja seda puudutavatest tehnoloogilistest valikutest.

1.1 Olemasolevad lahendused

1.1.1 Virtual Programming Lab

Virtual Programming Lab (VPL) [2] on Moodle'isse integreeritav pistikprogramm, mis võimaldab hallata ja automaattestida tudengite esitatud programmeerimisülesannete lahendusi. See on ka rakendus, mis on praegu Tartu Ülikoolis kasutusel ja mille puudujääkide lahendamise soovist tekkiski DeepMOOC platvormi loomise idee.

VPL dokumentatsioonist [3, 4] järeldub, et esitatud koodijuppide käivitamiseks kasutatakse serveripoolset teenust nimega VPL-Jail-System. Isoleeritud keskkond saavutatakse failisüsteemi duplitseerimisega ettenähtud kausta serveris ning muude Linuxi kernelis saadaval olevate isoleerimisvõtetega. Seega ei kasutata ei konteinereid ega muul viisil virtualiseerimist. See-eest soovitab manuaal keskkonna jooksutamiseks kasutada kas eraldiseisvat masinat või virtuaalmasinat, mis tuleb Moodle pluginas serverina registreerida.

VPL-Jail-System ise käitatakse selles masinas tavalise Linuxi teenusena. Selleks, et lisada uue programmeerimiskeele tuge, tuleb serveri masinasse paigaldada selle keele kompileerimiseks/interpreteerimiseks vajalikud komponendid ning siis nende poole pöörduda õppejõu koostatavas skriptis [3].

Autor järeldab, et kuigi VPL arendajad soovitavad kasutada serveri jaoks virtuaalmasinat, võimaldaks näiteks mõne selles oleva turvanõrkuse ärakasutamine ohtu seada serveritarkvara enda. DeepMOOC platvorm püüab seda vältida sellega, et ükski platvormi komponent ise testkeskkonnas ei jookse, vaid seda monitooritakse eemalt. Lisaks on testimise konteineri eluiga lühike (oleneb seatud ajalimiidist) ja iga uue testide komplekti puhul alustatakse nii-öelda puhtalt lehelt. Et VPLis tuleb kõigi toetatavate keelte komponendid koos vajalike teekidega paigaldada ühte keskkonda, võib see pikemas perspektiivis tekitada nendevahelisi konflikte ja üleliigset vaeva nende haldamisel.

1.1.2 Artemis

Artemis on interaktiivse õppe platvorm, mis peale programmeerimisülesannete automaatkontrolli toetab ka muid ülesandeliike, näiteks modelleerimist või testiküsimusi, ning sisaldab muuhulgas ka eksamite läbiviimise võimalust ja plagiaadituvastajat [5]. Siiski keskendub autor siinkohal ainult sellele, kuidas jooksutatakse tudengite koodi programmeerimisülesannete automaatkontrolli juures.

Artemis kasutab tudengite esitatud tööde jooksutamiseks Dockeri konteinereid, mis on loodud iga toetatud keele jaoks eraldi [6]. Seega on Artemis palju sarnasem planeeritavale DeepMOOC platvormile kui VPL.

Artemises on automaatkontroll realiseeritud versioonihaldussüsteemi Atlassian Bitbucket kaudu. Õppejõud loob graafilist kasutajaliidest kasutades Git repositooriumi ning lisab oma testkoodi. Repositooriumi juurde lisatakse vastavalt õppejõu tehtud valikutele skript Atlassiani Bamboo serveri konveieril koodi ja testide jooksutamiseks. Sealjuures kasutatakse Artemise poolt loodud Dockeri konteinereid, mille sees käivitatakse tudengi lahendus ja õppejõu testkood. [7]

Rakenduse arendajad on loonud Intellij arenduskeskkonnale plugina nimega Orion. Selle abil saab tudeng mainitud repositooriumi alla laadida ning seejärel lisada koodi testimiseks. Testimise lõppedes teavitab plugin tudengit tulemustest. [8]

DeepMOOC platvorm püüab olla võimalikult iseseisev, mistõttu selle arenduses Atlassiani või mõne muu sarnase teenusepakkuja servereid kasutada ei saa. Konveier ja võimalik versioonihaldussüsteem tuleb ise algusest lõpuni implementeerida. Siiski on võimalik kasutada ideid, mis puudutavad Artemises lähtekoodis Dockeri konteinerite enda ülesehitust.

Lisaks automaatkontrollile pakub Artemis ka staatilist koodianalüüsi [7], mis on ka DeepMOOC platvormi üks põhieesmärke, kuid Artemise analüüsitav puudutab pigem üldist koodi stiili või üritatakse tuvastada koodis esinevaid vigu kasutades näiteks SpotBugs rakendust. DeepMOOC pakub ülesande koostajale oluliselt rohkem tegutsemisruumi, mis tähendab, et Artemise pakutavast saab minna veel sammu edasi ning teostada ka ülesandespetsiifilist koodianalüüsi. Näiteks rekursiooniülesannete jaoks on võimalik luua keskkond, mis kontrollib, kas tudeng ikka on oma lahenduses rekursiooni kasutanud või sisaldab kood hoopis lubamatuid

tsükleid. Siiski eeldab taoliste kontrollide püstitamine analüüsiloogika kirjutamist ülesande koostaja enda poolt, kuid platvorm leidlikkusele piiranguid ei sea.

1.2 Tehnoloogilised valikud

1.2.1 Konteinerid ja Kubernetes

Nii platvormi enda kui ka tudengi ja õppejõu koodi käitatakse Dockeri konteinerites [9], mida omakorda haldab Kubernetes [10].

Konteinerid on kergekaaluline viis rakenduse või selle komponentide isoleerimiseks muust süsteemist. Erinevalt tavalisest virtuaalmasinast ei virtualiseerita konteinerite puhul operatsioonisüsteemi tuuma, vaid ainult sellest kõrgemal asuvaid kasutajataseme komponente. Sellest tulenev kergekaalulisus võimaldab korraga jooksutada paljusid konteinereid, mis jagavad küll kogu süsteemi ressursse, kuid tagavad siiski rakenduse loogilise isoleerituse ja sõltumatuse muust süsteemist. [11]

Konteineri tõmmis (*image*) on sisult nii-öelda juhend, kuidas mingit konteinerit ehitada. See koosneb kihtidest, kus iga kiht on omaette tõmmis, mis põhineb sellele eelnevatel kihtidel. Seega toimub uute tõmmiste loomine lisades olemasolevatele tõmmistele uusi kihte. [9]

Konteinerite eesmärk DeepMOOC platvormi tagarakenduses on tudengite esitatavate programmide nii turvakaalutlustel kui ka ressursside kasutamise mõttes isoleerimine. Lisaks võimaldavad konteinerid valmis seada igale toetatavale programmeerimiskeelele või nende eri versioonidele sobiva kasutajataseme keskkonna, ilma et need omavahel kattuks ja sellega serveris probleeme põhjustaks.

Põhjus, miks otsustati DeepMOOC platvormis kasutada esitatud koodi isoleerimiseks just konteinereid seisnebki nende väga suures paindlikkuses, aga samas ka kasutamise lihtsuses. Platvormile uue programmeerimiskeele toe lisamiseks ei ole vaja programmi muuta. Piisab lihtsalt sellele keelele vastava konteineri tõmmise loomisest, mida jooksutatakse nagu igat teist konteineritki ning ka suhtluses isoleeritud keskkonna ja tagarakenduse põhiloogika vahel ei ole sellest tulenevalt mitte mingisuguseid muutusi. Ainuke muudatus seisneb konveierile antavas parameetris, et millist konteineri tõmmist antud töö jaoks kasutada.

Kubernetes on oma olemuselt üsnagi laiahaardeline ja keerukas konteinerite orkestreerimise tehnoloogia, mistõttu seletatakse siin töös ainult mõningaid elementaarsemaid põhimõisteid, mis on vajalikud DeepMOOCi konveieri tööpõhimõttest arusaamiseks.

Lühidalt öeldes pakub Kubernetes raamistikku konteinertehnoloogiale rajatud rakenduse jooksutamiseks ja haldamiseks. Kubernetese klaster koosneb kogumist masinatest (*nodes*), mis käitavad rakenduse konteineritest komponente. Igale komponendile saab määrata kindla oleku, kasutades üksust nimega *deployment*. See näiteks määrab, mitu koopiat komponendist jooksma peaks. Kubernetes püüab alati hoida komponendi olekut sellisena, nagu see *deployment* failis määratletud on, ning kui näiteks mõni konteiner krahhib või peaks tekkima rike mõnes *node* masinas, siis proovib Kubernetes alati soovitud olekut taastada. *Deployment*i juurde lisatakse sageli *service*-nimeline võrguüksus, mille eesmärk on luua kõigile komponendi koopiatele ühene ligipääsuliides. Siis jaotab Kubernetes märkamatult koormust koopiate vahel ning süsteemi läbilaskevõime suurendamiseks saab lihtsalt komponendi koopiate arvu tõsta. [10]

Kubernetese klastri väikseimat üksust nimetatakse *pod*iks. See on sisuliselt abstraktsioon, mis ümbritseb ühte või mitut konteinerit, ning seob need ühtseks tervikuks, mis käitub Kubernetese klastris ühe üksusena [12]. DeepMOOC platvormis kasutatakse tudengi ja õppejõu koodi jooksutamiseks justnimelt *pod*i, mille sees jookseb vastavale programmeerimiskeelele sobilik konteiner. See annab konveierile jooksva koodi üle suure kontrolli. Näiteks on võimalik turvariskide minimeerimiseks sundida võõrast koodi jooksutav *pod* käivituma ainult selleks ette nähtud masinal ning rangelt piiratud ressurssidega [12, 13]. *Pod*i seisundit saab ka vastava teegiga tarkvaraliselt monitoorida ning vajadusel sekkuda.

Kubernetese kasutuselevõtt DeepMOOC platvormis oli planeerimisfaasis vaidlusalune teema. Sisuliselt saaks saavutada kogu soovitud funktsionaalsuse ka lihtsalt tavalist Dockerit kasutades. Selle eeliseks on asjaolu, et Kubernetese ülesseadmine ja haldamine on infrastruktuuri poole pealt üsna keerukas ja aeganõudev, kuid ainult Dockerile ehitatud lahenduse jooksutamine on võrreldes Kubernetesega haldust vähenõudev[2]. Põhiline argument, miks ikkagi Kubernetese kasuks otsustati seisneb selles, et see võimaldab tulevikku silmas pidades platvormi kerge vaevaga suuremaks skaleerida, et kasvava koormusega toime tulla. Arvestades, et DeepMOOCi võib hakata korraga kasutama mitme kursuse jagu tudengeid on koormusega toimetulek vägagi oluline. Autori enda seisukoht on, et kuigi Kubernetes on keerukam, on selle

[2] Põhineb arendusmeeskonna liikmete enda kogemusel.

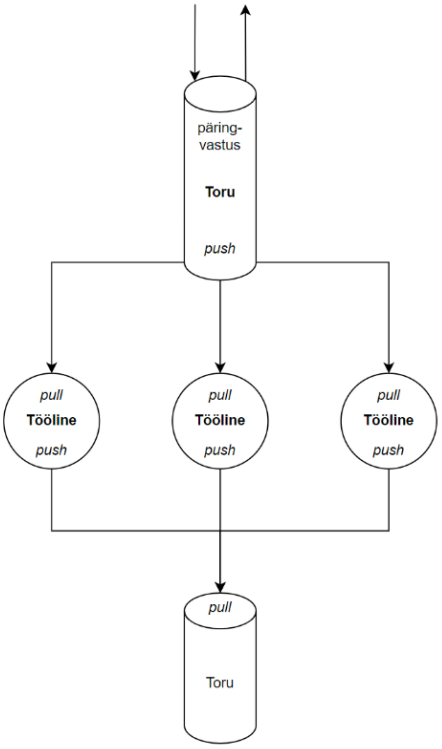
kasutuselevõtt kohe arenduse alguses investeering projekti tulevikku, mis tasub ära, kui projekti enda keerukus kasvab, see hakkab sisaldama rohkem võimalusi ning kaasab rohkem kasutajaid. Arvestades asjaolu, et konveier on väga tugevalt seotud konteinerite programmilise haldusega nõuaks tulevikus selle Kubernetesele ülekolimine suure hulga koodi ümberkirjutamist, kuna Kubernetese ja Dockeriga suhtlust pidavad teegid on täiesti erinevad.

1.2.2 Programmeerimiskeel ja teegid

DeepMOOC platvormi tagarakenduse ja sealhulgas ka konveieri programmeerimiskeeleks on Go, mis on moodne ning disaini poolest lihtsuse poole püüdlev Google'i arendatud keel. Go on loodud pidades silmas pilve ja mikroteenuseid [14], millega sarnaneb ka DeepMOOCi ülesehitus.

Go programmeerimiskeelele on olemas kõikvõimalikud teegid, mida DeepMOOC tagarakenduses vaja läheb, eelkõige Kubernetest ja Dockerit puudutav. Kusjuures mõlemad tehnoloogiad on ka ise Go keeles kirjutatud [14].

Kubernetese klastriga suhtlemiseks leidub Go keelele ametlik teek nimega Client-go. Seda teeki kasutab konveier klastrisse uue *pod*i loomiseks, kus jookseb esitatud kood, ning selle seisundi jooksvaks monitoorimiseks ja tekkinud logide kopeerimiseks, mille põhjal otsustatakse koodi testimise edukus.



Joonis 1. Sõnumiedastus konveieris

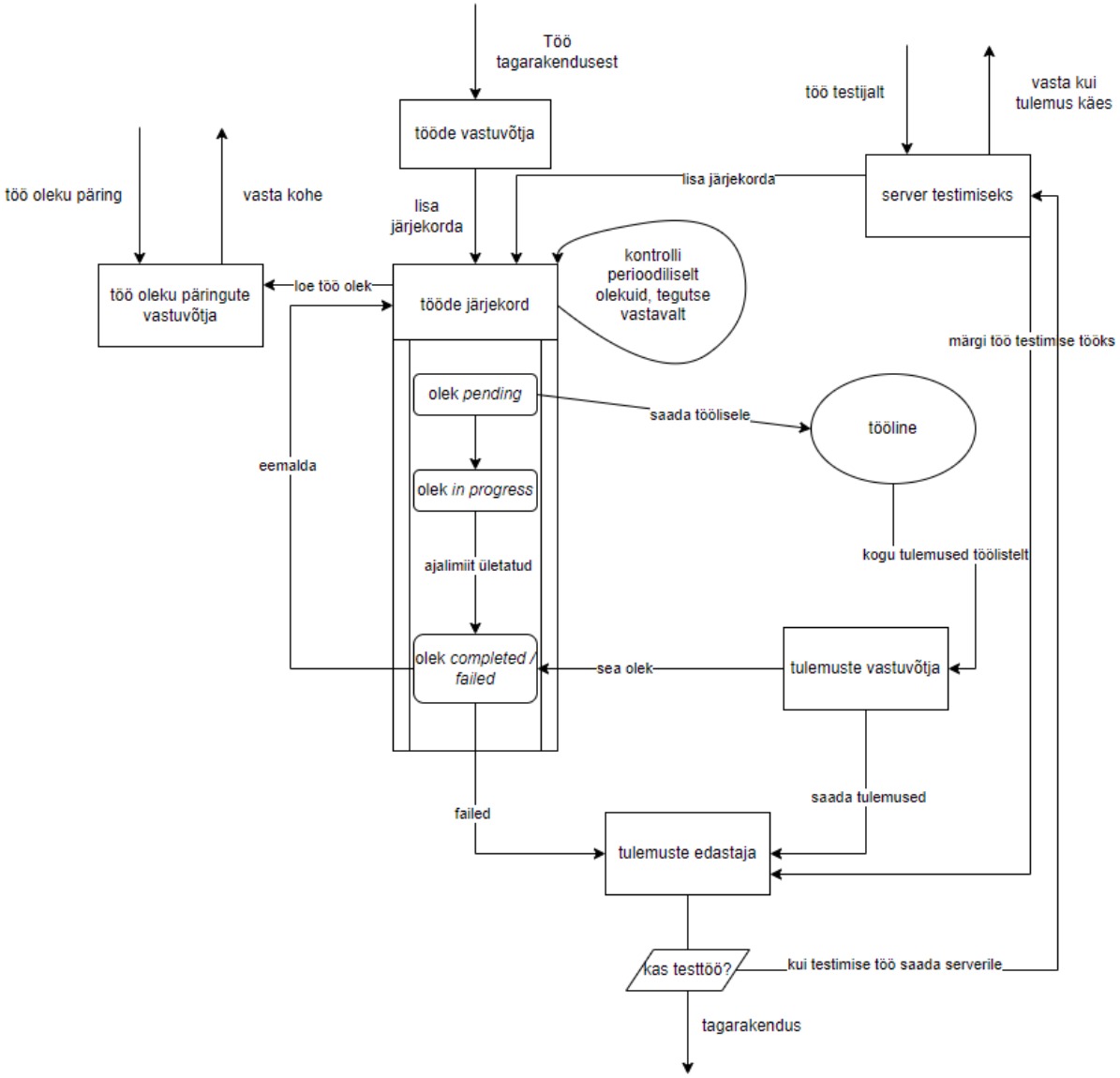
Tagarakenduse komponentide vaheliseks sõnumiedastuseks on kasutusel teek nimega Mangos, mis on laiemalt levinud Nanomsg teegi implementatsioon Go keelele [15]. See pakub nutikaid sokleid, mis tähendab, et ühenduste haldamine toimub suures osas teegi enda poolt ning arendaja saab rohkem keskenduda idee enda teostusele. Nanomsg pakub mitmeid laialt levinud sõnumiedastusviise [16] ning DeepMOOC platvormi konveier kasutab neist kahte (joonisel 1):

- Klassikaline päring-vastus, mis on kasutusel tööde vastuvõtmisel konveierisse.
- *Pipeline* või teise nimega *push-pull* muster, mis on ühesuunaline suhtlusviis, kus vastuvõtjad tarbivad saadetavaid sõnumeid vastavalt oma valmisolekule.

Alternatiividena olid kaalumisel ka näiteks gRPC ja Websocket suhtlusprotokollid, kuid Nanomsg/Mangos osutus valituks oma lihtsuse, kiiruse ja paindlikkuse tõttu. Suhtlus DeepMOOCi mõnede tagarakenduse komponentide võib olla üsna keerukas ja Nanomsg võimaldab seda seadistada oma äranägemise järgi sobilikumaks kui mõned muud protokollid.

1.3 Konveieri tööpõhimõte

Konveier koosneb kahest põhikomponendist: toru (*pipeline*) ja tööline (*worker*). Esimese ülesanne on võtta vastu sissetulevaid töid, neid võrdses mahus töölistele laiali jagada, tööde staatust uuendada ja raporteerida ning mõne lõppedes edastada selle tulemus.



Joonis 2. Toru ülesehitus

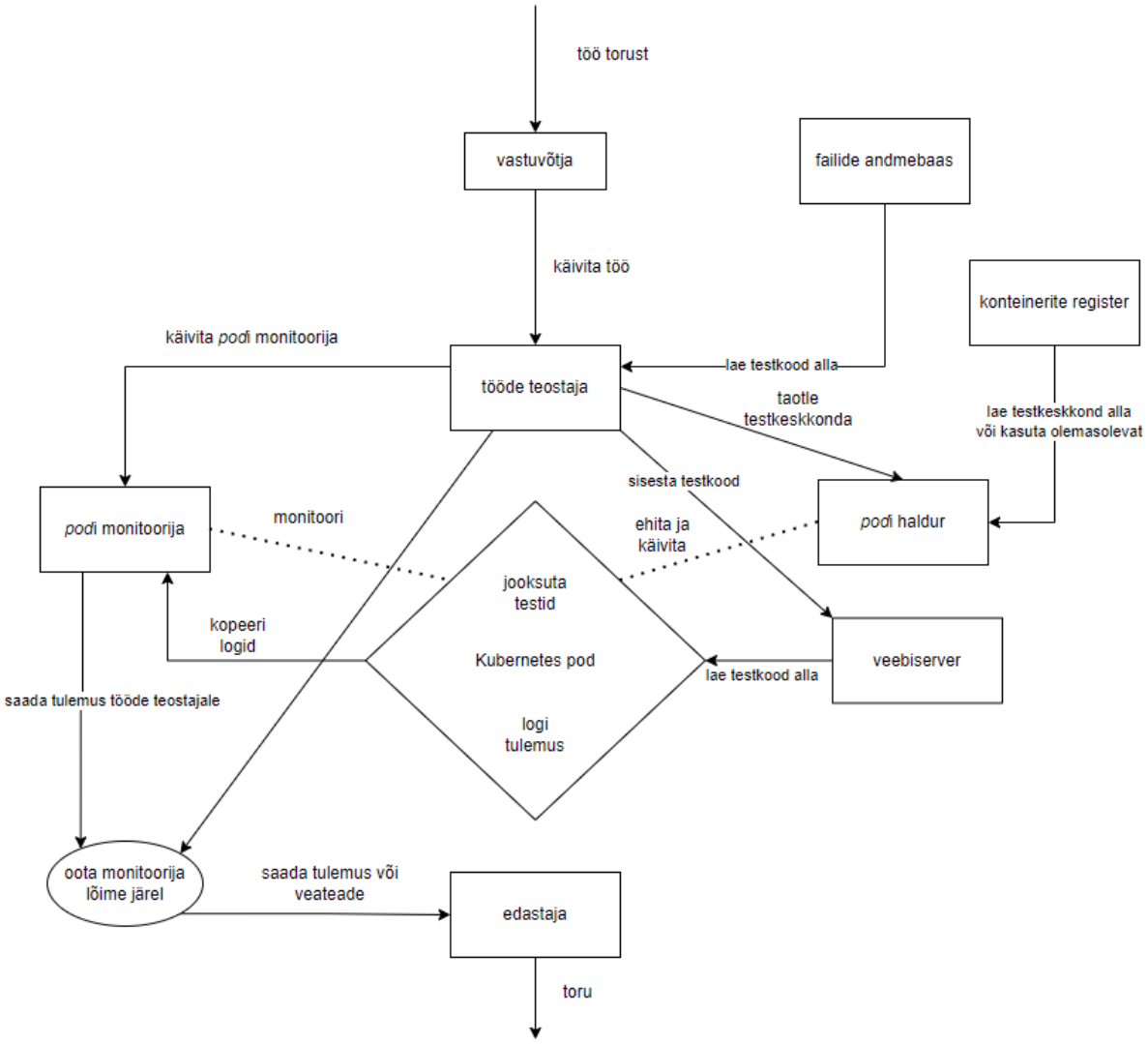
Joonisel 2 on toodud skeem toru sisemisest toimimisest. See on osa toru dokumentatsioonist ning sisaldab koodis endas olevat terminoloogiat, et tulevastel arendajatel oleks lihtsam programmi mõista. Valmis lahendus toetab lisaks skeemil toodule ka eri tüüpi töölisi ning ebaõnnestunud tööde teatud tingimustel uuesti saatmist, aga need on selguse huvides välja jäetud. Torus jookseb 5 erinevat lõime:

- Tööde vastuvõtja kuulab määratud pordil tagarakendusest tulevaid tööde nõudeid. Töö saabudes lisab selle tööde nimekirja.
- Teise lõime ülesanne on monitoorida tööde nimekirjas olevaid töid. Igale tööle on omistatud selle olekut kirjeldav muutuja, mille põhjal monitooriv lõim samme teostab. Näiteks kui olekuks on *pending*, siis üritatakse seda saata *push-pull* sõnumiedastusviisi kasutades töölistele ja õnnestumise korral märgitakse olekuks *in progress*. Olekuid on rohkem, kui joonisel toodud. Need lisaolekud on mõeldud haldamaks tööde uuesti saatmist ja parimal juhul läheb neid vaja harva.
- Kolmas lõim (tulemuste vastuvõtja) kuulab töölistelt tulevaid tööde tulemusi ning sõnumis oleva töö ID põhjal viib need kokku tööde nimekirjas olevate kirjetega. Õnnestunud töö tulemus saadetakse kohe edasi tagarakendusele, vastasel juhul märgitakse töö olekuks *failed* ning siis tegeleb sellega edasi monitooriv lõim, mis võib olenevalt veast otsustada selle uuesti saata või siis mitte.
- Neljas lõim (töö oleku päringute vastuvõtja) on mõeldud tagarakendusele, kui viimane peaks tahtma pärida mingi kindla töö olekut. Vastuseks saadetakse olek, mis on kirjas tööde nimekirjas.
- Viies lõim on tavaline http server, mis on mõeldud testimise otstarbeks. Sinna päringuid tehes saab saata süsteemi töid ning saada vastuseks töö tulemus.

Tööline tegeleb korraga ainult ühe tööga, mistõttu eksisteerib neid mitmeid, et koormust jaotada. Skeem töölise ülesehitusest on toodud joonisel 3, millel on välja toodud teenuse toimimise sammud:

1. Tööde vastuvõtja ootab sissetulevaid töid ning mõne saabudes saadetakse see töö teostamise ahelasse, mis sooritab järgnevad sammud.
2. Kõigepealt laetakse alla töö sõnumis viidatud zip-fail tudengi ja õppejõu koodiga, mis pakitakse lahti spetsiaalsesse kausta, mis on samas *pod*is jooksva veebiserveri käsutuses.

3. Siis käivitatakse testkeskkonda haldav koodi osa (*pod*i haldur), mis laeb sõnumis viidatud asukohast alla käsil oleva töö testkeskkonna Dockeri tõmmise. Kui see pole esimene kord, mil konkreetset keskkonda vaja läheb, siis kasutatakse mõistagi juba olemasolevat kohalikku tõmmist. Peale seda käivitatakse Kubernetese API kaudu testimiseks mõeldud *pod*, mis laeb töölise veebiserverist alla testimise koodi, käivitab selle ning väljastab tulemuse oma konsoolile.
4. Samal ajal käivitab tööde teostaja uue lõime (*pod*i monitoorija), mille ülesanne on monitoorida testkeskkonna *pod*i ning vastavalt seal toimuvatele sündmustele väljastada, kas tulemuse logi või siis veateade.
5. Pärast testimise lõppu testkeskkond ja testkood kustutatakse, edastatakse tulemus tagasi torule ning jäädakse ootama uut tööd.



Joonis 3. Töölise ülesehitus

2. Tööprotsess

Selles peatükis antakse ülevaade konveieri arendusprotsessist endast, kasutatud töövahenditest ning ideedest, mis viisid lahendusteni. Kuna programmeerimine oma olemuselt ongi probleemide lahendamine, siis ei laskuta tarkvara iga viimse detaili teostuseni vaid puudutatakse pigem olulisemaid tehnilisi väljakutseid, mille lahendamine oli protsess omaette.

Tarkvara loomiseks valis autor JetBrainsi arenduskeskkonna IntelliJ IDEA Ultimate, kuhu sai lisada Go keele toe vastava plugina installimisega. Valiku põhjuseks oli peamiselt varasem kogemus mainitud tarkvaraga ja hea Go keele tugi, mis tuli kasuks näiteks vigade otsimisel ja testide koostamisel. Lisaks sellele annavad IntelliJ'le installitavad programmeerimiskeelte pluginad sama tootja väiksemate arenduskeskkondadega (nagu näiteks Goland või Pycharm) samaväärse keele toe [17], mis tuli konveieri arendamisel kasuks, kuna testimiseks tuli luua eri keeltele mõeldud keskkondi, mida pluginate kasutamisega sai teha samast arenduskeskkonnast.

Kuna tööline suhtleb testimise *pod*ide loomiseks vahetult Kubernetese klastri endaga, siis tuli mõistagi tarkvara arendamiseks seda Kubernetese klastris käitada. Selle jaoks installeeris autor Minikube nimelise tööriista [18], mis võimaldab kerge vaevaga jooksutada kohalikus arvutis Kubernetese klastrit, ilma et oleks vaja vaeva näha täiemõõdulise Kubernetese ülesseadmisega.

Kubernetese *pod*ide ja muude ressursside haldamiseks kasutatakse kubectl nime kandvat käsurea tööriista. Et tarkvara arendamiseks oli seda vaja pidevalt ning vajaminevad käsud olid pikad ja lohisevad, installeeris autor veel täiendavalt k9s-nimelise programmi, mis loob mugava konsoolipõhise liidese klastri haldamiseks [19]. Selle abil saab mõne klahvivajutusega *pod*e näiteks kustutada, neisse siseneda ja logisid kuvada, hoides nii kokku aega, mis muidu kuluks pikkade käskude kirjutamiseks.

Konveier kasutab oma töös välist failide andmebaasi ning konteinerite registrit. Kuna neid arendamise ajal veel olemas ei olnud, siis kasutas autor nende asendamiseks tavalist Ubuntu Server [20] virtuaalmasinat, kus käitas kahes Dockeri konteineris klassikalist veebiserverit failide andmebaasi asendamiseks ning Dockeri enda pakutavat konteinerite registri tõmmist, mille abil saab kerge vaevaga luua küll ebaturvalise, aga testimiseks piisava kohaliku registri [21].

2.1 Testimine

Peale igat koodimuudatust teenuse konteineri uuesti ehitamine ja selle Minikube'is tööle panemine on tüütu ja ajamahukas protsess. Seetõttu otsustas autor arendamisele läheneda pigem ühiktestide põhiselt. See tähendab, et peale uue koodijupi loomist, selle asemel, et lisatud funktsionaalsust kogu süsteemis proovida, kirjutas autor hoopis seda katva testi, mis kontrollis, kas lisatud kood toimis ootuspäraselt. Lisaks ajakokkuhoiule suurendab selline lähenemine ka tarkvara kvaliteeti, kuna suurem osa koodist on lõpplahenduse valmides automaattestitud.

Et kogu ehitatud süsteemis on väga palju üksteisest sõltumatuid ja eriilmelisi komponente, siis selle töökindluse tagamiseks ainult ühiktestidest ei piisanud. Töö käigus teostas autor ka konveieri koormustestimist selle jaoks koostatud skriptiga. Näiteks, kui Minikube'i klastris jooksis kolm töölist, siis koormas autor süsteemi üle, saates samal hetkel torusse 20 tööd. Koormustestimine on oluline ka selle poolest, et DeepMOOC platvormi puhul võivad suure koormusega ajahetked olla tulevikus väga sagedased, näiteks kui vahetult enne programmeerimise kodutöö tähtaega suur hulk tudengeid selle korraga esitab.

Vigu, mis sellisel viisil välja tuli, leidus mitmeid, kuid valdavalt olid need seotud sellega, kuidas tööline testkeskkonna loomisega toime tuleb. Näiteks juhtus, et uus töö tuli peale enne, kui Kubernetes oli jõudnud vana testkeskkonna ära kustutada, mistõttu andis viimane veateate. Selle lahendas autor niiviisi, et lisas testkeskkonna *pod*i nimesse käsil oleva töö ID, et Kubernetes neil vahet teeks ja saaks uue keskkonna käivitada juba enne vana *pod*i kustutamise lõpuni viimist.

Teine klass vigu puudutas tudengi koodi tegutsemist testkeskkonnas. Veahalduses tuli vahet teha, kas testimise *pod* krahhis mingi süsteemi enda vea tõttu (näiteks ei saanud tõmmist alla laadida) või põhjustas selle tudengi kood. Seda infot läks vaja torus töö uuesti saatmisel, sest mittekompileeruva või krahhiva koodiga tööd pole mõtet uuesti proovida. Autor lähenes taolistele probleemidele valmistades ette rea testkomplekte koodiesituste eri stsenaariumite jaoks:

- Kood, mis on korrektne ja läbib kõik õppejõu ühiktestid.
- Kood, kus mõni õppejõu test läbi ei lähe.
- Esitused, kus krahhib tudengi kood või õppejõu kood.
- Lahendus, mis võtab liiga kaua aega, ületades töölises seatud ajalimiidi.

- Lahendus, mis satub lõpmatusse tsüklisse ning kirjutab testkeskkonna konsoolile tohutus koguses logisid.

Enim vigu tõid esile kaks viimast stsenaariumi. Näiteks selgus, et ajalimiiti ületades ei õnnestunud mõnikord kopeerida testkeskkonnast tulemuse logisid, kuna töölises olid monitooriva lõime ja Kubernetese *pod*ile antud ajalimiidi võrdsed, mistõttu jõudis viimane hakata *pod*i kustutama enne, kui logid said kopeeritud. Selle lahendus oli lihtne – piisas monitoorivale lõimele mõne sekundi võrra lühema ajalimiidi andmisest. Lõpmatu tsükkel põhjustas kaks probleemi: Kubernetesel kulus sellise *pod*i kustutamiseks kaua aega ning kuna logisid tekkis väga palju ei õnnestunud nende välja lugemine. Esimese probleemi lahendas autor, lisades töölises Kubernetese API kustutamise käsu juurde parameetri (GracePeriodSeconds: 0), mis pani klastri testimise *pod*i jõuga sulgema, selle viisakalt kustutamise asemel. Et tegu on ühekordse testkeskkonnaga, pole vahet, kuidas see sulgub. Teise juhtumi lahendamiseks seadistas autor logide kopeerimisele limiidi 0,1 MiB, mille täitudes edasisi logisid ignoreeritakse.

Platvormi kiire toimimine on hea kasutajakogemuse seisukohalt tähtis, mistõttu teostas autor konveieris ka ajamõõtmisi. Selleks midagi uut programmi lisada polnud vaja, piisas mõõdetavate teenuste logide lugemisest, kuna nende juurde lisatakse automaatselt ajatempel. Selgus, et kogu protsessi kõige ajamahukam osa on töölises testkeskkonna käivitamine, millele kulub umbes 80% ajast. Ilma optimeerimisi tegemata kulus ühe java testide komplektile vastuse saamiseks umbes 6-7 sekundit. Kui torusse saadeti selline töö, mis kasutas tühja testkeskkonda ehk failidega seal midagi ei tehtud, siis kulus sama ahela läbimiseks umbes 5 sekundit. Seega sõltub ajakulu suuresti sellest, mismoodi on testkeskkond ülesande koostajal üles seatud. Kuluvat aega õnnestus märkimisväärselt vähendada, kui tavalise java Dockeri tõmmise asemel kasutati Alpine Linuxil põhinevat tõmmist. Tegemist on väga kergekaalulise tõmmisega, mille suurus võib arendaja sõnul piirduda lausa 8 MB'ga [22]. Sellist testkeskkonda kasutades kulus samale testide komplektile vastuse saamiseks 3-4 sekundit ja tühjalt jooksutatuna 1-2 sekundit. Seega soovitab autor edaspidi Dockerfile'ide kirjutamisel võimalusel alati kasutada Alpine Linuxil põhinevat tõmmist. Samuti tuleb mainida, et need ajamõõtmised teostati autori arendusmasinal, mis oli Minikube'st, testimise virtuaalmasinast ja muudest rakendustest tingituna suure koormuse all, ning tõenäoliselt kulub platvormi jooksutavatel serveritel vähem aega.

2.2 Lahendusteni jõudmine

2.2.1 Konveieri ülesehitus

Töö alguses oli vaja täpselt paika panna kogu süsteemi üldine arhitektuur, mis võimaldaks kõiki platvormile esitatud nõudeid täita ning oleks tulevikus hõlpsasti laiendatav. Kaalumisel oli eri variante, mida selles alapeatükis põgusalt käsitletakse.

Esimene idee, mida autor kaalus, kujutab endast arhitektuuri, kus otseselt töölisi ei olegi, vaid toru tegeleb ise testkonteinerite loomisega ning kõige muu selle juurde käivaga. Sellisel üles–ehitusel ilmnes aga mitmeid puudujääke. Esiteks pole see kuigi veakindel. Tuleb arvestada, et toru võib korraga hallata kümneid esitatud lahendusi ning kui mingil põhjusel peaks tekkima viga, nii et toru krahhib, siis läheksid kõik need lahendused kaduma. Seega, kui liigutada testkeskkonna haldus eraldi mikroteenusesse, vähendab see süsteemi keerukust ning võimaldab vigade tekkimisel näiteks töö uuesti saata või ka veast tagarakendusele teada anda. Lisaks muudab taoline struktuur süsteemi paindlikumaks ning võimaldab tulevikus lisada uusi funktsionaalsusi eri tüüpi töölisi kasutades. Seda analüüsitakse täpsemalt järgnevates alampeatükkides.

Eraldiseisev tööline loob juba mainitud eelistele ka võimaluse süsteemi lihtsasti skaleerida. Selleks tuleb lihtsalt Kuberneteses tõsta tööliste koopiate arvu ning ilma muid muudatusi tegemata, süsteemi läbilaskevõime suureneb. Skaleerimise ja eri tüüpi tööliste puhul võib süsteemi haldajal tekkida soov mingisugust gruppi töölisi käivitada kindlal masinal. Seda saab Kuberneteses lihtsasti teha lisades vastava võtme teenuse *deployment*i kirjeldusse. Siin tekib aga probleem: kui töölisi mitte kasutada, siis tuleks testkeskkonna masin määrata torus tarkvaraliselt, mis oleks väga keerukas, kui mitte võimatu, sest ilmselt vajaks see iga töö juurde parameetri lisamist, mis kirjeldaks, millisel masinal see konkreetne töö jooksma peab. Tööliste puhul seda probleemi ei esine, kuna testimise *pod*i loomisel saab selle tarkvaraliselt määrata käivituma alati samal masinal, kus ka tööline ise on, viimase asukoha saab süsteemi haldaja seadistada vastavat tüüpi töölist kirjeldavas *deployment* failis.

Teine arhitektuuri puudutav dilemma käis töölise ja testkeskkonna omavahelise paigutuse üle. Esialgne idee oli need tihedamalt kokku siduda, käivitades töölise *pod*i sees kaks konteinerit, üks põhiloogika jaoks ja teine tudengi ja õppejõu koodi jooksutamiseks. Autor oleks sellise struktuuri juurde ka jäänud, kuid selgus, et Kuberneteses pole võimalik *pod*i muuta (ehk töölisele teist konteinerit lisada), ilma seda *pod*i uuesti loomata [12]. See omakorda kaotanuks ära DeepMOOC platvormi põhiliseks müügiargumendiks oleva paindlikkuse, kuna iga toetatava

keele jaoks tulnuks töös hoida eraldi töölist, mille sees on testitavale keelele mõeldud keskkond. Valituks osutunud arhitektuuris seda probleemi ei ole, kuna testkeskkond luuakse iga töö saabudes uuesti. Niiviisi saab eri keeltes olevaid töid käitada sama töölise peal läbisegi ehk tööline on universaalne ning ei sõltu programmeerimiskeelest, mille teste see teostab. See muudab kogu süsteemi haldamise ja ka uute keelte lisamise oluliselt lihtsamaks, kuna ainus, mis uue keele toe lisamiseks teha tuleb, on vastavat keskkonda kirjeldava Dockerfile'i kirjutamine, vastasel juhul tuleks teha süsteemis struktuurilisi muutusi.

2.2.2 Eri tüüpi töölised

DeepMOOC platvorm püüab olla võimalikult laiahaardeline ning mitte piirduda ainult ühiktestide käivitamisega. Sisuliselt võimaldab praegune konveieri struktuur teostada testkeskkonnas mistahes tegevust ning seda piirab ainult ülesande püstitaja leidlikkus, siiski võib ette tulla olukordi, kus on vaja kasutada tavapärasest erinevat töölist ja testimise *pod*i konfiguratsiooni. Üheks selliseks on näiteks testitava koodi käitusaja mõõtmine, mis on ka platvormi seisukohalt oluline, kuna aja põhjal on plaanis luua edetabeleid.

Aja mõõtmise puhul tuleb tagada, et testkeskkonnal olevad ressursid, nagu näiteks protsessori tuumade arv ja mälu maht, oleks iga lahenduse testimise puhul võrdsed. Seda on võimalik saavutada, lisades vastavad parameetrid testkeskkonna *pod*i Kubernetese konfiguratsioonifaili. Siin võib tekkida küsimus, et milleks on antud juhul vaja spetsiaalset tüüpi töölist, kui sellise töö saabudes saaks lihtsalt viimasele ette anda parameetri, mis paneb selle kasutama ajamõõtmiseks mõeldud teist testimise *pod*i konfiguratsioonifaili? Vastava toe lisamisel oli selline idee ka autoril, kuid paraku tekitas selline lähenemine teatud konflikte varasemate disainiotsustega. Nimelt on antud kasutusjuht justnimelt see, kus väga suure tõenäosusega on süsteemi haldajal soov käitada ajamõõtmiseks mõeldud testkeskkondi eraldi masinal, vähendades sellega teiste samaaegselt jooksvate tööde mõju mõõtmistulemusele. Kuna testkeskkonna loob tööline ning varasemalt oli tehtud otsus, et keerukuse vältimiseks tuleb mitte määrata testkeskkonna masinat tarkvaraliselt, siis ongi ainus lahendus siduda iga testimise *pod*i konfiguratsioon kindla töölise tüübiga, millele saab siis soovitud masina käsitsi määrata.

Võimaldades eri tüüpi töölisi, saab tulevikus platvormi edasi arendada, näiteks lisades sinna selliste tööde teostamise võimalusi, millede puhul pole testkeskkonna loomine vajalik ehk mõistlik on kasutada täiesti teistsuguse ülesehitusega töölist.

Uut tüüpi tööliste grupi lisamiseks tuleb süsteemi haldajal lisada selle grupi kirjeldus toru konfiguratsioonifaili, luua vastav töölise *deployment* ning (vajadusel) testimise *pod*i konfiguratsioon ja esimene klastris käivitada. Selleks, et uut tüüpi teenus toruga ühendust saaks, tuleb konfiguratsioonifailis määratud pordid avada ka toru *service*'is. Protsess võib tunduda keeruline, kuid autor leiab, et uut tüüpi tööliste lisamine on harv tegevus ning sisuliselt osa tarkvara arendusprotsessist. Kuna niiviisi on platvormi võimalik täiendada kõikvõimalike lisadega, millest paljud ka selle töö skoobist välja jäävad, siis on täpsem protsessi kirjeldus välja toodud platvormi dokumentatsioonis, mis on lisatud konveieri lähtekoodi juurde.

2.2.3 Testkeskkonna Dockerfile'i ülesehitus

Üks suuremaid tehnilisi probleeme, mis lahendada tuli, seisnes selles, et kuidas sisestada tudengi esitatud kood testimise *pod*i ning see käivitada nii, et protsessi käigus poleks vaja konteinerit uuesti ehitada. Viimase teostamine on väga ajamahukas ja palju süsteemiressursse nõudev, mistõttu muudaks see kogu platvormi toimimise aeglaseks ja kulukaks.

Nagu eelnevalt mainitud, Dockerfile koosneb reast käskudest, mida konteineri tõmmise ehitamiseks jooksutatakse ülevalt alla, kusjuures iga käsk loob omaette tõmmise kihi. Kui mõnes kihis toimub muudatus, siis tuleb kogu tõmmise taasloomiseks uuesti ehitada kõik kihid, mis sellele järgnevad. Erandiks on viimane, tavaliselt CMD nime kandev käsk, mis jookseb siis, kui konteiner käivitatakse, ehk viimane kiht lisandub alles konteineri käivitumisel, mitte ehitamisel [23]. Seega selleks, et tõmmist poleks vaja esitatud koodi lisamise järel iga kord uuesti ehitada, on ainuke käsk, mille raames saab koodi lisamine, vajadusel kompileerimine ja käivitamine toimuda, just see kõige viimane. Ehk sisuliselt tuleb luua testkeskkonda koodifailidest sõltumatu skript, mis mainitud tegevused konteineri käivitumisel teostab.

Esimene lahendusena katsetas autor töölisele ja testimiskeskkonnale ühise Kubernetese volüümi (*volume*) loomist. Volüüme on Kubeneteses mitut eri tüüpi ning need on viis andmete salvestamiseks ja jagamiseks eri teenuste vahel [24]. Idee seisnes selles, et tööline kopeerib failid volüümiga ühendatud kausta ning siis saaks testkeskkond oma skriptis neile ligi vastava haakepunkti (*mountpoint*) kaudu. See lähenemine tekitas paraku rea täiendavaid probleeme.

Esiteks suurendab see võimalikku ründepinda. Tuleb arvestada, et testimise *pod*'il jookseb väline kood, mis võib üritada testkeskkonna konteinerist välja murda ning sooritada mis iganes pahatahtlikku tegevust süsteemi seeläbi kahjustades. Dockeri konteinerite tehnoloogias on juba

varasemalt leitud mitmeid turvaauke, mis taolist tegevust võimaldaksid [25]. Seega tuleb kasuks, kui eeldada, et seni leidmata nõrkusi eksisteerib veel ning mida vähem testkeskkonna puhul Kubernetese pakutavaid võimalusi kasutada, seda väiksem on tõenäosus, et ründaja leiab mõnest neist nõrkuse, mille abil konteinerist välja murda. Volüümi kasutades oleks testkeskkonna ja töölise failisüsteemid omavahel seotud ning seetõttu leiab autor, et sellise konstruktsiooni kasutamine vähendab turvalisust, seda enam, et töölisel on erinevalt teistest *pod*idest õigus klastrit ennast muuta (testimise *pod*ide loomiseks).

Kaalumisel oli ka teine sarnane lahendus: NFS (Network File System) kasutamine testkoodi ülekandmiseks kahe *pod*'i vahel. NFS on protokoll, mis on võimaldab võrku luua andmehoidla, mille saab haakida eri masinate failisüsteemide külge, nii et need kõik samadele failidele ligi pääsevad [26]. Idee oli luua klastrisse täiesti eraldi *pod* NFS serveriga, millega siis nii tööline kui ka testkeskkond ühenduda saaksid. Taolise struktuuriga saab mikroteenuste vahel lihtsasti andmeid jagada ning ka Kubernetes võimaldab lisada vastava nfs-tüüpi volüümi *pod*i külge [24], kuid DeepMOOC kasutusjuhu puhul lisab see olulise turvanõrkuse. Nimelt ei õnnestunud töö autoril käivitada NFS serveriga *pod*i, ilma vastavale konteinerile privilegeeritud õigusi andmata. Ilmselt vajab NFS oma võrguga seotud tegevusteks suuremat ligipääsu masina kernelile, mis annab sisuliselt juurkasutaja lähedased õigused hostsüsteemis [27], mis on tõenäoliselt veel hullem turvarisk kui lihtsalt Kubernetese volüümide kasutamine.

Lahendus, mis lõpuks kasutusse läks, loob töölise *pod*i teise Dockeri konteineri. Üldjuhul jookseb ühes Kubernetese *pod*is üks konteiner, kuid Kubernetes võimaldab neid ka seal rohkem käivitada. Samas *pod*is olevad konteinerid jagavad võrku ning saavad omavahel suhelda üle *localhost*i [12], seega on eri teenuste vahelist võrgusuhtlust lihtne teostada. Lisaks sellele pakub Kubernetes volüümi tüüpi nimega *emptyDir*. See on ruum mida *pod*is olevad konteinerid saavad omavahel jagada, kuid mis on *pod*i loomisel alati tühi ning ka kustutatakse koos *pod*iga, kus see paikneb [24].

Niisiis valmis töölises toimub tudengi ja õppejõu koodi testkeskkonda üle kandmine järgmiselt: *pod*is eksisteerib kaks konteinerit, kus esimene sisaldab töölise põhiloogikat ennast ning teine on tavaline Apache veebiserver, mille ainus ülesanne on ühe kausta failide üle võrgu kättesaadavaks tegemine. Kui testkood on alla laetud, kopeeritakse see kindlaks määratud kausta, mis on *emptyDir* abil jagatud nii põhikonteineri kui veebiserveri kausta vahel. Testkeskkonna konteineri loomisel käivitub skript, mis siis töölise *pod*ist vastavad failid alla laeb. Siin võib tek-

kida küsimus, et kuidas testkeskkond teab, milline tööline ta käivitas ehk millise *pod*i IP-aadressilt testkood alla laadida? Selle lahendamiseks kasutas autor ära asjaolu, et testimise *pod* luuakse töölises tarkvaraliselt, mis võimaldabki sinna kergesti lisada keskkonnamuutuja töölise *pod*i enda IP-aadressiga.

Testide tulemus loetakse testkeskkonnast välja, kasutades töölises Kubernetese APIt *pod*i logide kopeerimiseks. Taoline lähenemine on autori hinnangul kõige turvalisem, kuna ei nõua testkeskkonnast väljaviiva suhtluskanali avamist, mida pahatahtlikud kasutajad võivad proovida ära kasutada. Selleks, et tulemuse logi saaks süsteemis lihtsasti ringi liigutada ja lugeda, ilma et tekiks probleeme näiteks reavahetustega, kodeeritakse see välja lugemisel base64 kujule.

Järgnevalt on toodud näidis Dockerfile, mis seadistab java keskkonna Junit 5 testide jooksutamiseks. Kõigepealt laetakse alla ja installitakse testide jooksutamiseks vajalikud programmid (wget ja Junit 5 jar-fail). Skirpt koosneb kolmest käsust: esimene (wget) laeb testimise failid töölisest alla, teine (javac) kompileerib programmi ning viimane (java) käivitab testid. Et väljund näeks ilus välja, on viimasele lisatud täiendavaid parameetreid.

```
FROM openjdk:19-jdk-alpine

RUN apk update && apk add bash wget
WORKDIR /app


RUN wget --quiet https://repo1.maven.org/maven2/org/junit/platform/junit-
platform-console-standalone/1.8.2/junit-platform-console-standalone-1.8.2.jar
RUN mkdir out


RUN echo "#!/bin/bash" > start.sh
RUN echo "wget -r -nH -np --reject='index.html*' --quiet \$TESTFILES_DL_URL \
&& javac -d out -cp out:junit-platform-console-standalone-1.8.2.jar *.java \
&& java -jar junit-platform-console-standalone-1.8.2.jar --class-path out --
scan-class-path --disable-banner --disable-ansi-colors --details-theme=ascii"
>> start.sh
RUN chmod 777 start.sh
CMD ./start.sh
```

Seega selleks, et luua Dockerfile'i DeepMOOC platvormi jaoks peab see täitma järgmiseid nõudeid:

- Dockerfile'is tuleb keskkonda lisada skript, mis oskab konteineri käivitudes koodifailid töölisest alla laadida ning testid käivitada. Ühtegi faili, mis võib iga esituse puhul olla erinev, eelnevalt keskkonda lisada ei saa.
- Failide allalaadimiseks peab skript kasutama URLi, mis on saadaval TESTFILES_DL_URL keskkonnamuutujas.
- Dockerfile peab lõppema CMD instruktsiooniga, kus käivitatakse mainitud skript. Testide tulemuse peab skript väljastama konteineri konsoolile. Midagi muud konsoolile väljastada ei tohi.

3. Tulemus

Töö tulemusena valmis DeepMOOC platvormi ühte põhiülesannetest ehk tudengi ja õppejõu koodi jooksutamist võimaldav tarkvaraline konveier, mille lähtekood on ka tööga kaasa pandud, täpsemalt on seda kirjeldatud töö lisades. Valminud lahendus sisaldab järgmist:

- tööde vastuvõtmine torusse ning nende töölistele laiali jagamine
- konkreetse töö oleku raporteerimine tagarakendusele
- Kubernetese klastrisse ajutise testkeskkonna loomine ja selle haldamine
- koodi käitamine testkeskkonnas ning tulemuse väljastamine
- võimalus luua uusi testkeskkondi, kirjutades neid kirjeldavaid Dockerfile'e
- võimalus laiendada toru võimekust uut tüüpi tööliste lisamisega
- testimisotstarbeline http server torusse tööde saatmiseks
- ühiktestid konveieri koodis

Paraku teiste platvormi komponentide arendus selle töö valmimise hetkeks nii kaugele jõudnud ei olnud, et saanuks neid konveieriga siduda ning süsteemi üles seada. Siiski leiab autor, et nii mõnedki selle töö raames ületatud tehnilised väljakutsed nagu näiteks komponentide vaheline sõnumiedastus ja Kubernetese seadistust puudutavad otsused, toetavad kindlasti edasist arendust. Lisaks sellele võimaldab konveieri olemasolu näiteks tagarakendust arendades seda mugavamalt testida, saates torusse töid ning saades vastuseid.

Et loodud tarkvara polnud veel võimalik reaalsetele serveritele jooksma panna, jäid tõenäoliselt töö valmides lahendamata sellega kaasneda võivad tehnilised üksikasjad, nagu näiteks põhjalikum konveierit puudutavate Kubernetese komponentide seadistus. Lisaks sellele, kuna arendamise ajal puudusid nii failide andmebaas kui ka konteinerite register, siis ei sisalda valminud konveier konfiguratsiooni, mis on vajalik nendele teenustele ligipääsu saamiseks. Seetõttu on tulevikus vajalik konveieri koodi ja seda puudutavat Kubernetese konfiguratsiooni täiendada, et töölisel oleks võimalik mainitud teenustest vajalikke andmeid alla laadida.

Autor leiab, et valminud osa platvormist vajab teiste komponentide valmides ka põhjalikumat integratsiooni ja süsteemistestimist, tagamaks, et konveieris leiduvad võimalikud tarkvaravead, mis võivad tuleneda näiteks teiste komponentide sisendist, mida autor ei saanud ette näha, saaksid kõrvaldatud.

Üks võimalik edasiarendus, milleni selle töö raames küll ei jõutud, on võimalus testkeskkonnast välja kopeerida ka muud tüüpi infot kui ainult logisid. On täiesti mõeldav kasutusjuht, kus testide tulemuseks on mingisugune fail – näiteks peab testitav kood looma graafi kujutise PNG-vormingus. Siis tekib vajadus see testkeskkonnast välja kopeerida ning tulemusena kuvada. Sellele on võimalik lähenda sarnaselt logide kopeerimisele, kasutades Kubernetese teeki Client-go.

Kokkuvõte

Bakalaureusetöö eesmärk oli luua DeepMOOC platvormile tarkvaraline konveier, mis on suuteline teostama programmeerimisülesannete automaattestimist, võimaldades seejuures rohkemat, kui praegu kasutusel olev VPL. Kuna töö kirjutamise hetkel platvorm veel reaalses kasutuses ei ole, ei saa ka tuua võrdlust kumb lahendus tudengitele ja õppejõududele rohkem meeldib, kuid valminud konveier pakub võimaluse automaattestimine palju laialdasemalt ja mitmekesisemalt kasutusele võtta, kui teeb seda praegune lahendus VPL. Seetõttu usub töö autor, et DeepMOOC platvormil on tulevikus potentsiaali VPLiga tugevalt konkureerima hakata.

Töö teoreetilises osas anti ülevaade konveieris kasutusel olevatest tehnoloogiatest ning kirjeldati nende valiku põhjuseid. Lisaks seletati lahti konveieri komponentide: toru ja töölise ülesehitus ning tööpõhimõtted. Praktilises osas kirjeldati tööprotsessi ennast: kuidas eelnevalt kirjeldatud lahenduseni jõuti, millised ideed veel kaalumisel olid ning miks need valituks ei osutunud.

Autori hinnangul sai konveier oma funktsionaalsuselt valmis piisavas mahus, et selle saaks reaalselt kasutusele võtta, kuid integratsioon teiste platvormi komponentidega on veel puudulik, sest need ei olnud selle töö kirjutamise ajaks piisavas mahus valminud. Seetõttu on vajalik teha täiendavat arendustööd, tagamaks, et komponentide vaheline suhtlus toimiks ladusalt.

Viidatud kirjandus

[1] L.-C. Cheng, W. Li ja J. C. R. Tseng. Effects of an automated programming assessment system on the learning performances of experienced and novice learners. 2021. https://www.tandfonline.com/doi/full/10.1080/10494820.2021.2006237 (20.12.2021).

[2] VPL, the Virtual Programming lab for Moodle. https://vpl.dis.ulpgc.es/. (20.12.2021)

[3] Advanced features - Virtual Programming Lab for Moodle (VPL) 3.4.3+ documentation. https://vpl.dis.ulpgc.es/documentation/vpl-3.4.3+/advancedfeatures.html (20.12. 2021)

[4] VPL Jail System's documentation. https://vpl.dis.ulpgc.es/documentation/vpl-jail-system-2.7.0/index.html (20.12.2021)

[5] Artemis: Interactive Learning with Individual Feedback. https://docs.artemis.ase.in.tum.de (20.12.2021)

[6] Builds and Dependency Management - Artemis documentation. https://docs.artemis.ase.in.tum.de/dev/docker/ (20.12.2021)

[7] Programming Exercise - Artemis documentation. https://docs.artemis.ase.in.tum.de/user/exercises/programming/ (20.12.2021)

[8] Orion - Artemis documentation. https://docs.artemis.ase.in.tum.de/user/orion/ (20.12.2021)

[9] Docker overview. https://docs.docker.com/get-started/overview/ (26.03.2022)

[10] Kubernetes Documentation – Concepts. https://kubernetes.io/docs/concepts/ (27.03.2022)

[11] Google Cloud. What are containers? https://cloud.google.com/learn/what-are-containers (20.12.2021)

[12] Kubernetes Documentation – Pods. https://kubernetes.io/docs/concepts/workloads/pods/ (26.03.2022)

[13] Resource Management for Pods and Containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (26.03.2022)

[14] Go for Cloud & Network Services. https://go.dev/solutions/cloud. (27.03.2022)

[15] Mangos – Documentation. https://pkg.go.dev/go.nanomsg.org/mangos/v3#section-readme (02.05.2022)

[16] Nanomsg documentation. https://nanomsg.org/ (09.05.2022)

[17] IntelliJ IDEA overview. https://www.jetbrains.com/help/idea/discover-intellij-idea.html (09.05.2022)

[18] Minikube documentation. https://minikube.sigs.k8s.io/docs/ (02.05.2022)

[19] K9s - Kubernetes CLI To Manage Your Clusters In Style! https://k9scli.io/ (02.05.2022)

[20] Ubuntu Server Guide. https://ubuntu.com/server/docs (09.05.2022)

[21] Deploy a registry server - Docker documentation. https://docs.docker.com/registry/deploying/ (09.05.2022)

[22] Alpine Linux – About. https://www.alpinelinux.org/about/ (09.05.2022)

[23] Best practices for writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (02.05.2022)

[24] Volumes - Kubernetes documentation. https://kubernetes.io/docs/concepts/storage/volumes (02.05.2022)

[25] R. Yasrab. Mitigating Docker Security Issues. *arXiv: Cryptography and Security*. 2018, lk 3-4 (09.05.2022).

[26] IBM. Network File System – Documentation. https://www.ibm.com/docs/en/aix/7.1?topic=management-network-file-system (02.05.2022)

[27] Pod Security Policy - Kubernetes documentation. https://kubernetes.io/docs/concepts/security/pod-security-policy/#privileged (02.05.2022)

Lisa - lähtekood

Tööga on kaasas zip-fail „lahtekood.zip“, mis sisaldab konveieri lähtekoodi ja *readme*-kujul olevat dokumentatsiooni, mis on eelkõige suunatud tulevastele arendajatele.

Kaustas „pipe“ asub toru lähtekood, selle ehitamiseks vajalik Dockerfile, arendamisel kasutatud toru konfiguratsioonifail „config.json“ ning kaks *markdown*-vormingus olevat dokumentatsioonifaili:

- „README.md“ fail sisaldab kiirülevaadet torust, mis on mõeldud olema abiks tagarakenduse arendamisel, ning juhendit, kuidas lisada uut tüüpi töölisi.
- „SETUP.md“ fail sisaldab juhendit, kuidas luua oma arvutisse konveieri arendamiseks sobilikku arenduskeskkonda.

Kaustas „worker“ on toodud töölise lähtekood, mille juures on samuti *readme*-fail. See sisaldab ülevaadet töölisest ning juhendit uute testkeskkondade loomiseks. Arhiivifailis sisaldub ka kaust „manifests“, milles on konveieri komponentide Kubernetese konfiguratsioonifailid.

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Andre Anijärv**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **DeepMOOC platvormile tarkvarakonveieri arendamine**,

   mille juhendajad on Tõnis Hendrik Hlebnikov ja Ahti Põder,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace, kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autori õiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Andre Anijärv

10.05.2022