TARTU ÜLIKOOL
Arvutiteaduse instituut
informaatika õppekava

Mark-Eerik Kodar

FaaS pilveplatvormide võrdlus

Bakalaureusetöö (9 EAP)

Juhendaja(d): Pelle Jakovits

Tartu 2019

FaaS pilveplatvormide võrdlus

Lühikokkuvõte:

Üha enam arendatakse tarkvara pilves ning viimastel aastatel on tekkinud uus võimalus hostida pilves tervete rakenduste asemel ainult funktsioone. Selliseid lahendusi kutsutakse FaaS (Functions-as-a-Service) platvormideks ning käesolev töö annab ülevaate olemasolevatest FaaS pilveplatvormidest ning võrdleb kahe kasutusloo põhjal AWS Lambda ja IBM OpenWhisk platvorme. Selle töö eesmärk on uurida praktiliste võrdluste abil seni avaldatud FaaSi kohta avaldatud artiklites välja toodud kriitikat ning üldiselt uurida, kui lihtne on seada üles funktsioone väljatoodud pilveplatvormidel. Töö tulemused kinnitasid, et FaaS platvormide hetke suurimateks probleemideks on külmkäivitusest tingitud mõjud. Lisaks näitasid tulemused, et andmevahetus teiste pilveteenustega võib mõjutada funktsiooni latentsust.

Võtmesõnad:

FaaS, IaaS, serverivaba arhitektuur, pilvearvutus

CERCS: P170 - Arvutiteadus, arvanalüüs, süsteemid, kontroll

Abstract:

More and more software is developed on the cloud and in the recent years a new functionality has been developed that enables developers to host standalone functions in the cloud instead of monolithic applications. These solutions are named FaaS (Functions-as-a-Service) platforms and this thesis will give an overview of different FaaS cloud platforms and compares AWS Lambda and IBM OpenWhisk based on two use cases. The goal of this thesis is to determine whether the problems described in the published articles are still present and how easy it is to set up functions on the cloud. The results of the thesis confirmed that currently the greatest problems come from cold starting the function. In addition to that the tests showed that interaction with other cloud services might affect the function's latency.

Keywords:

FaaS, IaaS, serverless, cloud computing

CERCS: P170 - Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus

Viimaste aastate jooksul on aina enam keskendutud sellele, et pilveteenuste platvormidel hostida rakendusi või teenused, kus teenusepakkuja annab võimaluse kasutada nende poolt pakutavat riistvara pilves. Sellise arhitektuuri stiili nimetuseks on infrastruktuur teenusena (ingl *Infrastructure-as-a-Service – IaaS*) ning selline arhitektuur võimaldas arendajatel hallata kasutatavat riistvara pilves. See tähendas ka seda, et vajadusel oli arendajatel võimalik lihtsalt riistvara juurde rentida, kui nende rakendusel kasutajate arv suurenes või oli vaja rohkem funktsionaalsuseid implementeerida, mis nõudsid rohkem arvutusressurssi [1]. Kuid viimasel ajal on populaarsust kogunud mikroteenuste kasutamine rakenduste tarkvaralises arhitektuuris. See tähendab seda, et selle asemel, et hostida monoliitset rakendust pilveplatvormil jaotatakse rakenduse tervik programmeerimisel üksteisest sõltumatuteks teenusteks, kus iga teenus on vastutav rakenduse mingi funktsionaalsuse eest ning on täielikult eraldatud teistest funktsionaalsustest. Sellisest tarkvaraarhitektuurist on välja arenenud omakorda veel suurem jaotuseaste ehk kus iga rakenduse funktsioon on eraldiseisev teenus pilvekonteinerites ning ei sõltu teistest rakenduse komponentidest. Konteiner on virtualiseerimise meetod, kus rakendus paigutatakse eraldatud keskkonda koos operatsioonisüsteemi kerneliga ilma, et oleks vajadus luua operatsioonisüsteemiga virtuaalmasinat. [1]

Taoline ülesehituse peamiseks eeliseks võrreldes IaaS platvormiga on see, et arendajad vähem tarkvarakeskkonda seadistama ja ka haldama. Üldjuhul hoolitseb platvorm virtuaalsete konteinerite loomise, skaleeruvuse ning optimeerimise eest ning kasutaja peamiseks rolliks jääb funktsionaalsuse implementeerimine ja testimine. Sellise arhitektuuri nimi on funktsioonid teenusena (*Functions-as-a-Service - FaaS*) või laiemalt serverivaba arhitektuur ning seda lahendust pakuvad mitmed pilveteenuste platvormid nagu Amazon (AWS Lambda[2]), Google (Google Cloud Functions[3]), Microsoft (Azure Functions[4]) ja IBM (IBM OpenWhisk[5]). Lisaks neile on ka mitmeid vabavaralisi lahendusi nagu OpenFaas ja OpenWhisk.

[1] https://akit.cyber.ee/term/8651-containerization
[2] https://aws.amazon.com/lambda/
[3] https://cloud.google.com/functions/
[4] https://azure.microsoft.com/en-us/services/functions/
[5] https://www.ibm.com/cloud/functions

Kuigi FaaS lahendusel on mitmeid varasemalt mainitud eeliseid võrreldes varasemate pilvetehnoloogiatega nagu näiteks IaaS, siis tegu on siiski väga uue tehnoloogiaga ja lisaks sellele on viimasel ajal on avaldatud kriitilisi teadusartikleid nende platvormide praeguste kasutusvõimaluste koht. Näiteks väidetakse [2], et mitmel juhul on FaaS lahenduste kasutamine ebamõistlik ja selle jõudlus mitmel kasutusjuhtumil küsitav ning ka sellest tulenevalt võib ka FaaS platvormide kallim olla kui mõne IaaS platvormi kasutamine. Samuti on olnud artiklid väga tihti keskendunud AWS Lambda platvormile ning ei ole väga rõhku pööratud erinevate platvormide võrdlusele. Selles töös uuritakse kirjeldatud AWS Lambda probleeme täpsemalt ning uurida nende probleemide olemasolu ka teisel pilveteenuse platvormil, mis on baseeruv avatud tarkvaral (IBM). Lisaks sellele on see töö oluline demonstreerimaks, kuidas ja kui lihtne või keeruline on luua erinevaid rakendusi FaaS platvormidele. FaaS platvormide üheks eeliseks on see, et see lubab alustavatel ettevõtetel seada enda rakendus kiirelt üles ning testida seda. Selline mudel lubab neil kiirelt prototüüpida erinevaid lahendusi ning selle abil leida endale pikas perspektiivis parim lahendus. Kuna erinevaid kasutusjuhtumeid on palju, siis see töö keskendub kahele erinevale kasutusjuhtumile, milleks on registeerimisvorm ning pilditöötlus.

Järgmises peatükis antakse ülevaade serverivabast arhitektuurist, tuntumate serverivabade platvormide teenusepakkujate poolt pakutavatest FaaS teenustest, milleks on AWS Lambda, Google Cloud Functions, IBM Cloud Functions ning vabavaralistest lahendustest OpenFaas ja OpenWhisk. Kolmandas peatükis seatakse kaks kasutuslugu üles Amazoni ja IBM platvormidel ning antakse ülevaade testimiseks vajalike skriptide loomiseks. Neljandas peatükis analüüsitakse testide tulemusi ning sellele järgneb viimases peatükis kokkuvõte.

2. Taustainfo

See peatükk annab esiteks ülevaate serverivabast arhitektuurist üldiselt ning seejärel on ülevaade levinumatest FaaS pilveteenuste pakkujatest ning vabavaralisest FaaS lahendusest.

2.1. Serverivaba arhitektuur

Serverivabale arhitektuurile [3] ei ole ühtset definitsiooni, kuid see mõiste hõlmab laias laastus kahte domeeni, millest esimene on mobiilne *backend* teenusena ehk BaaS (*backend-as-a-service)*. See mõiste hõlmab rakendusi, mis on suures osas või täielikult sõltuvad pilveteenusepakkujate rakendustest ja teenustest, et hallata serveripoolset loogikat. Teine domeen on sellised rakendused, kus serveripoolne loogika on kirjutatud arendaja poolt, aga seda koodi käivitatakse olekuta konteinerites, mis on sündmuste põhised ja hallatavad täielikud kolmanda osapoole poolt. Sellist domeeni nimetatakse FaaSiks (*Functions-as-a-Service*). FaaSi [4] põhimõte on väga lihtne ning tegelikkuses pärineb tavalisest programmeerimisõpikust, kus kirjeldatakse funktsioone puhtalt kui struktuuri, mis võtab vastu sisendi ja tagastab väljundi, kusjuures traditsiooniline programm koosneb just nendest funktsioonidest. FaaS platvormid pakuvad seda kõike lihtsalt pilvepõhiselt ehk pilvepõhistest funktsioonidest moodustatakse pilvepõhine tervik.

Serverivaba arhitektuuri mõistega [5] käib kaasas ka selle nimest tulenev vale arusaam, et reaalset füüsilist serverit ei olegi vaja. Tegelikult serverit on siiski vaja ja see on füüsiliselt kuskil maailmas olemas, kuid arendaja peab vähem nendega tegelema, sest serverivaba arhitektuuri [3] korral liiguvad funktsionaalsused, mis olid reeglina hallatavad keskse serveri poolt, kolmandate osapoolte kätte, kus iga funktsionaalsuse jaoks on eraldiseisev süsteemi osa. Selliseid funktsionaalsused võivad olla näiteks andmebaas, autentimine, sessiooni jälgimine ja skaleeruvus. See tähendab seda, et sellise süsteemi ülesehituse juures „eelistatakse koreograafiat orkestratsioonile“ (ingl *choreography over orchestration*). Selle eelisteks on see, et kui meil on iga süsteemi komponent eraldiseisev osa, siis neid on kergem tulevikus vajadusel muuta või uuendada. Teisalt tähendab see seda, et sellisel juhul on süsteemi seire ja silumine raskendatud, sest on väga raske süsteemikeskselt kuidagi teha.

Põhiline erinevus FaaSi ja IaaS platvormide vahel on see, et FaaS võimaldab jooksutada serveripoolset koodi ilma, et oleks vaja servereid hallata, sest FaaS on fundamentaalselt üles ehitatud sellele, et funktsioonid käivitamiseks vajaminevaid ressursse kasutatakse ainult siis, kui selle jaoks reaalne vajadus on ning makstakse selle kasutusaja ja ressursi eest, mida kasutati. PaaS teenusepakkuja puhul tuleb maksta fikseeritud tasu serverihaldamise eest.

See tähendab ka seda, et FaaSi puhul on võimalik neid ressursse dünaamiliselt juurde hankida, kuid PaaSi puhul seda ei ole võimalik otseselt ja automaatselt teha. Üheks suureks PaaSi eeliseks [5] on see, et kasutajatel on kontroll virtuaalmasinate riistvara üle ning saavad seeläbi optimeerida seda vastavalt enda rakenduste vajadustele. Teisalt on väga tõenäoline, et tulevikus [4] on see võimalus olemas ka FaaS platvormidel.

Iga FaaS [3] [5] platvorm on tegelikkuses sündmuste käsitlemise ja töötlemise platvorm, mis peab haldama kasutaja poolt defineeritud funktsioone ning neid vajadusel käivitama, kas süsteemisiseste sündmuste või HTTP päringute läbi. Seejärel peab süsteem kindlaks tegema millist funktsiooni käivitada, leidma selle käivitamiseks vajaliku instantsi või selle ise looma. Sellele instantsile saadetakse sündmus, kus funktsioon töötleb seda ning saadab kasutajale vastuse. Platvorm vastutab samal ajal ka selle eest, et kõike seda logitakse, et kasutaja saab tegelikult vastuse kätte ja lõpuks peab platvorm ka vastava instantsi sulgema, kui seda enam vaja ei ole.

FaaSi funktsioonide üheks nõrkuseks [3] on see, et neil ei ole püsivat lokaalset mälu, mis püsiks mitme funktsiooni väljakutse vahel ning kunagi ei saa kindel olla selles, et mingid kindlad muutujad funktsiooni väljakutsumisel säilivad, sest funktsiooni võidakse välja kutsuda täiesti teises funktsiooni instantsis. Sellest tulenevalt öeldakse, et FaaS funktsioonid on olekuta ning juhul, kui on vaja püsiandmeid kasutada, siis neid on vaja teha FaaSi funktsiooni väliselt. Seda võimaldavad andmebaas, sõnumiteenused või objektikogumid. Lisaks sellele limiteerib [5] FaaSi kasutamist see, et soovitud platvorm ei pruugi toetada keele uuemaid versioone või lisateeke. Samuti kipub FaaS teenusepakkujate puhul olema nii, et kõiki teisi teenused nagu andmete hoiustamine, logimine, autentimine on kõik seotud selle sama teenusepakkujaga, mis tähendab, et soovi korral võib teisele FaaS platvormile migreerumine raskendatud ning tekib oht jääda kinni ühe teenusepakkuja juurde (ingl *vendor lock-in*)

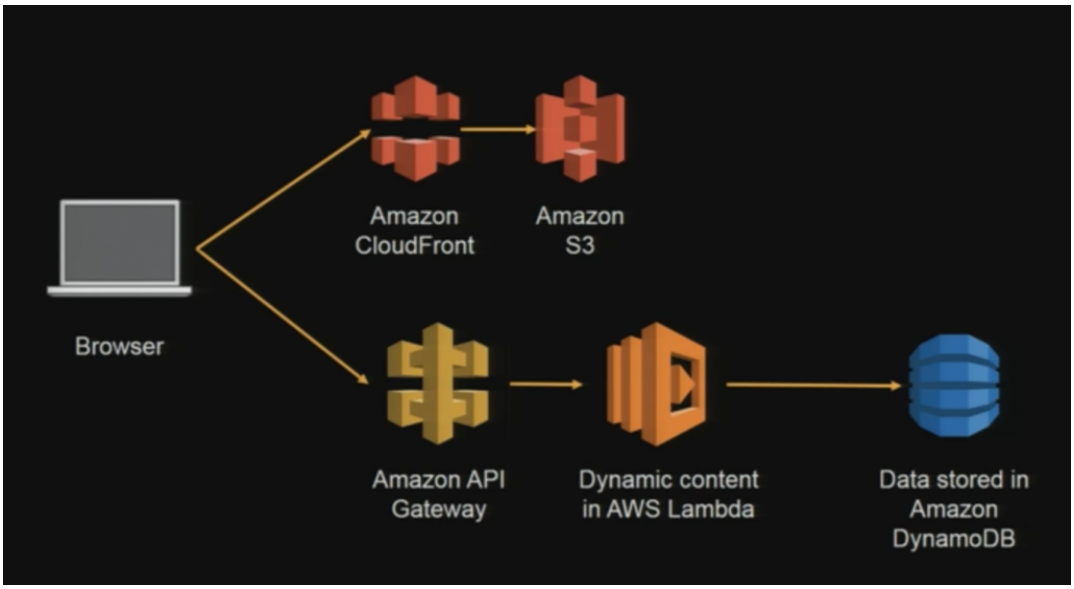
Üheks FaaSi nõrgimaks kohaks on funktsioonide nö külmkäivitamine (ingl *cold start*), mis tähendab, et juhul kui funktsiooni ei ole tükk aega kasutatud, siis selle esimese instantsi loomine võib võtta märkimisväärselt kauem aega võrreldes sellega, kui mõni funktsiooni instants on aktiivne funktsiooni väljakutsel. Külmkäivitamisel tekkiv latentsus sõltub mitmest asjast: programmeerimiskeel, väliste teekide arv, koodiridade arv, väliste ressursside vajadus jne. Õnneks on paljud nendest asjadest arendaja kontrollida ning seeläbi on võimalik vähendada külmkäivitamisel tekkivat latentsust. Teisalt ei pruugi olla külmkäivitamine probleem, kuid see sõltub suuresti rakenduse olemusest, sest kõrge

liiklusega rakenduste puhul on väga ebatõenäoline, et see võib tekkida, küll aga võib see probleemne olla väiksema liiklusega rakenduste puhul, kus näiteks kutsutakse funktsioone välja iga tunni tagant. Üheks alternatiiviks [2] väiksema liiklusega rakendustele on selliste funktsioonide üles seadmine, mis hoiab teisi funktsioone nö soojana.

2.1.1. AWS Lambda

Amazoni poolt pakutav FaaS platvorm kannab nime AWS Lambda [6]. Amazoni funktsioone annab välja kutsuda teiste AWS teenuste poolt nagu S3, DynamoDB, Kinesis, SNS, CloudWatch ning Lambda funktsioone on võimalik orkestreerida AWS Step Functions abil [7], mis võimaldab funktsioone seadistada jadana nii, et iga funktsioon võtab sisendiks temale eelneva funktsiooni väljundi. Sealjuures on iga samm jälgitav ja logitav ning funktsiooni väljakutsel tekkivaid probleeme üritatakse automaatselt lahendada proovides funktsiooni mitut korda automaatselt uuesti käivitada. AWS Lambda [8] toetab vaikimisi järgmisi programmeerimiskeeli: Node.js, Python, Ruby, Java, Go, .NET.

AWS Lambda [9] on hinnastatud vastavalt päringute arvule kõikide funktsioonide peale, kus üks päring on sündmuse teavitus, funktsiooni väljakutse või testimisel tehtud funktsioonide väljakutsed. Hinda mõjutab ka ajakulu, mida arvestatakse alates koodi käivitamise algusest kuni see tagastab midagi või vastav funktsioon lõpetatakse jõuga. Lisaks neile arvestatakse hinna hulka ka kasutatud mälu hulka. Amazoni platvorm pakub iga kuu tasuta miljon päringut ja 400 000 GB-sekundis arvutusressurssi. Nende ületamisel on teenus juba tasuline. Lisaks sellele on tasuline ka see, kui rakendus suhtleb teiste Amazoni teenustega nagu Amazon S3, kus hinnastamine on seotud andmebaasi lugemise ja kirjutamise arvuga. Üldiselt on iga arvutuse hinnaks 0.00001667 dollarit iga GB-sekundis kohta ja iga päringu hinnaks on 0.20 dollarit iga miljoni päringu kohta. Joonisel 1 on näidatud veebirakenduse näitestruktuur Amazoni pilveplatvormil, kus kasutatakse AWS Lambda funktsioone veebilehe dünaamilise sisu näitamiseks.



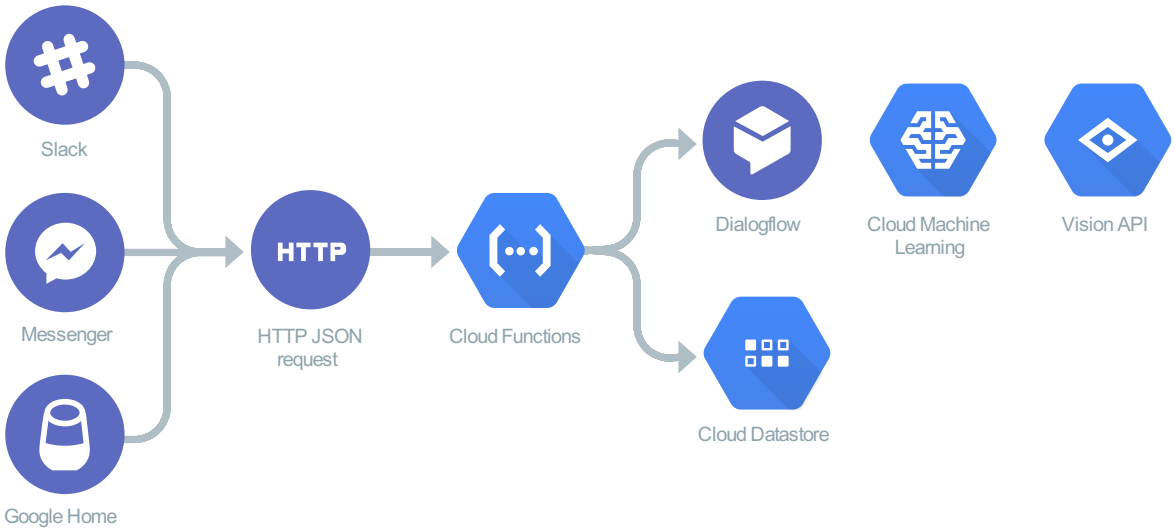
Joonis 1. Näiteskeem AWS Lambda kasutamise kohta. [6]

2.1.2. Google Cloud Functions

Google poolt pakutavaks FaaS lahenduseks on Google Cloud Functions [10] ning see toetab programmeerimiskeeltest hetkel ainult Node.js, Python ja Go käitusmootoreid. Sarnaselt teistele FaaS lahendustele on Google Cloud Functions teenuses võimalik funktsioone välja kutsuda üle REST API ja taustal toimuvate platvormisiseste sündmustega nagu näiteks mõnes Google Cloud Pub/Sub sõnumi või faili üleslaadimisel. Joonisel 2 on võimalik näha näiteskeemi Google Cloud Functionsi kasutamisest. Google Cloud Functions [2] oma olemuselt on üsnagi kerge, sest ta jagab suurema osa ülesannetest Google Firebase [11] platvormile, mis on mobiilse serveripoole rakendus. Viimane pakub logimist, autentimist, andmete hoiustamist ja testimist.

Sarnaselt Amazonile kasutab Google platvorm [12] hinnastamist kasutusmahu põhjal. Esimesed kaks miljonit funktsiooni väljakutset on tasuta ning üle selle on iga miljoni funktsiooni väljakutse hind 0.4 dollarit. Lisaks sellele on 400 000 GB-sekundis arvutusvõimsust tasuta kuid selle ületamisel on arvutusvõimsuse ning mälu kasutuse hind on 0.0000025 dollarit iga GB-sekundis kohta.

[6] https://aws.amazon.com/blogs/startups/introducing-the-startup-kit-serverless-workload/



Joonis 2. Näiteskeem Google Cloud platvormi kasutamiseks juturoboti näitel. [7]

2.1.3. IBM Cloud Functions

IBM Cloud Functions [13] on IBMi poolt pakutav FaaS lahendus, mille aluseks on võetud avatud lähtekoodiga Apache OpenWhisk, millest tulenevalt kasutatakse platvormi nimena ka IBM OpenWhiski. IBMi lahendus [14] on üles ehitatud selliselt, et kasutajal on võimalik luua tegevusi (ingl *action*), mis reageerivad kindlatele sündmustele. See tegevus võib olla puhtalt koodijupp või ka Dockeri kujutis.

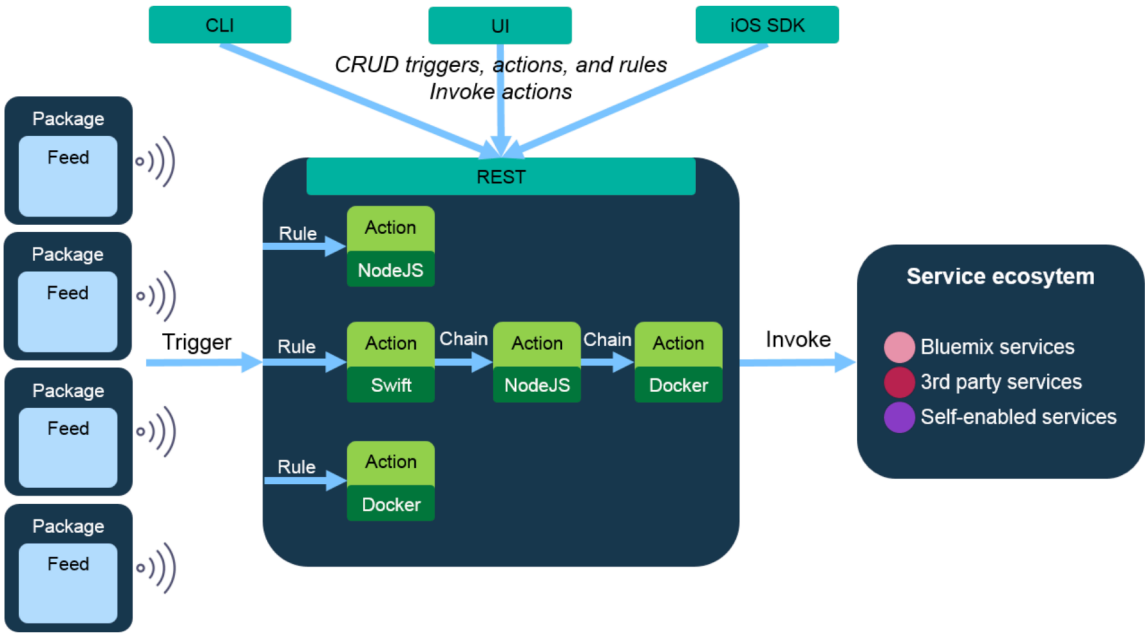
Tegevusi võib välja kutsuda läbi IBMi enda Cloud Functions API, CLI või iOSi SDK. Lisaks sellele on võimalik tegevusi käivitada sütikute (ingl *triggerite*) ja kolmanda osapoolte APIde kaudu. Platvorm võimaldab ka lisaks tegevused ritta panna ilma koodi kirjutama, moodustades tegevuste jada (ingl *sequence*), kus ühe tegevuse väljund on järgmise sisendiks. See võimaldab tegevusi taaskasutada ja vajadusel erinevalt kombineerida. Jada välja kutsumine käib sarnaselt üksikule tegevusele läbi REST API. Trigger tähendab seda, et kindlat tüüpi sündmusele reageeritakse kindlat moodi ning seda, mis moodi reageeritakse mingile sündmusele kirjeldavad reeglid (ingl *rule*).

Tegevuse väljakutsel pannakse käima kood, mis on selle kindla tegevusega seotud ning tagastatakse selle funktsiooni tulemus. Joonisel 3 on näha, mis moel on võimalik funktsioone välja kutsuda ning kuidas need suhtlevad teiste teenustega.

IBM on ainuke teenusepakkuja, kes pakub OpenWhiski protokolli peale loodud täisteenuselist platvormi. IBM on [14] ei sea piiranguid sellele, mis keeles peab olema see koodijupp kirjutatud, kui tegemist on Dockeri tõmmisega, kuid programmisiseselt on

[7] https://cloud.google.com/functions/use-cases/intelligent-applications

toetatud JavaScript, Swift, Java, Go, PHP, Python, .NET ja Ruby. Sündmustetootjatest toetatakse Github repositooriumit, Cloudant andmebaasi, IBM Message Hub, Mobile Push ja Periodic Alarm platvorme.



Joonis 3. Funktsiooni väljakutse IBM OpenWhisk platvormil.[8]

2.1.4. OpenFaas

Erinevalt eelnevalt mainitud FaaS platvormidest ei ole OpenFaaSi näol tegemist pilveplatvormiga, vaid OpenFaaS [15] on tasuta vabavaraline raamistik, mille abil on võimalik ehitada ja hostida serverivabu funktsioone. Raamistik toetab vaikimisi Docker Swarmi ja Kubernetest ning laseb funktsioone välja kutsuda üle HTTP päringute, käsurea kaudu või üle veebiliidese.

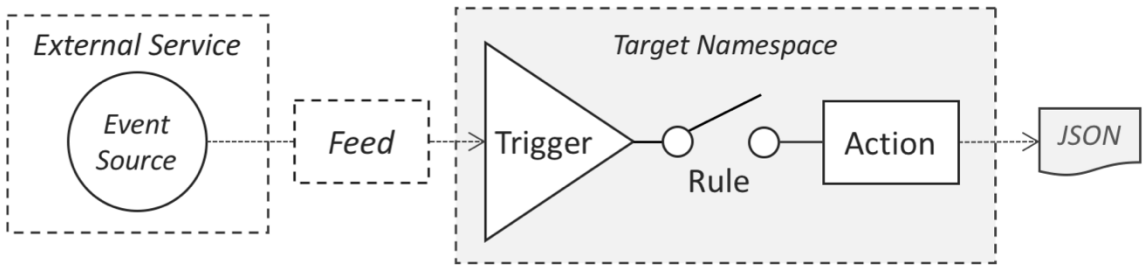
2.1.5. Apache OpenWhisk

Apache OpenWhisk on [16] vabavaraline serverivaba platvorm, mis suudab hallata infrastruktuuri erinevaid komponente kasutades Dockeri konteinereid. Platvorm toetab vaikimisi NodeJSi, Swifti, Javat, God, Pythonit, PHPd, Rubyt ja Ballerinat, kuid on võimalik kasutada ka teisi programmeerimiskeeli läbi Dockeri SDK. Lisaks sellele on Apache poolt loodav tarkvara kergesti ühilduv teiste platvormidega. Näiteks toetatakse Kafka sõnumiplatvormi, Cloudanti andmebaasi, Slacki suhtluskeskkonda, GitHubi ja JIRAt.

[8] https://console.bluemix.net/docs/openwhisk/openwhisk_about.html#openwhisk_about

Funktsioone [17] on võimalik välja kutsuda sünkroonselt, asünkroonselt või vastavalt kindlale ajavahemikule. Erinevalt teistest kommertsplatvormidest ei ole OpenWhiskis süsteemisiseseid sündmuseid, mille abil funktsiooni välja kutsuda, vaid funktsioone saab väliselt või luua mooduleid välistest teenustest, mis kuulavad sündmuseid.

Joonisel 4 on näha Apache OpenWhiski programmeerimismudelit. Kuna IBM OpenWhisk on põhinev Apache vabavaralisel versioonil, siis siin on samade terminitega kirjeldatud protsessi erinevaid osasid. Tegevused (ingl *actions*) on olekuta koodijupid, mis tegelevad rakenduse loogikaga vastavalt funktsiooni sisendile. Tegevusi saab välja kutsuda OpenWhisk REST API, OpenWhisk CLI, kasutaja poolt loodud API või automatseeritud sütikute abil (ingl *trigger*). Tegevusi saab samuti siduda tegevustejadaks, kus ühe funktsiononi sisendiks on teise väljund. Reeglid (ingl *rule)* määravad selle, et milline sütik on seotud millise tegevusega. Joonis 4 näitab üldist tegevuste skeemi funktsiooni väljakutsumisel.



Joonis 4. Sündmuste ahel funktsiooni väljakutsel.[9]

2.2.Seotud tööd

Seoses sellega, et FaaS ja serverivaba arhitektuur on küllaltki uus mõiste, siis väga palju artikleid sellel teemal avaldatud ei ole. Sõna *serverless* hakkas populaarsust koguma alles 2015. aastal. Hetkel avaldatud artiklite seas on põhiliselt keskendutud Amazoni poolt pakutud Lambda teenusele ning seetõttu on vähem kajastatud teised teenusepakkujad või on töödes keskendutud mingile kindlale kasutusloole. Samuti on üsna vähe võrreldud erinevaid FaaS platvorme üksteise vastu, kuid on võrreldud näiteks FaaS platvorme näiteks virtualiseerimisega. Seega on see töö vajalik eelkõige seetõttu, et sellist võrdlust ei ole otseselt tehtud ning juhul, kui on vaja valida mitme erineva FaaS teenusepakkuja vahel, siis võib otsuse tegemisel selle töö aluseks võtta.

[9] https://openwhisk.apache.org/documentation.html

Seni avaldatud töödest on enamik olnud ülevaatliku olemusega, mis kirjeldavad erinevad serverivaba arhitektuuri kasutusvõimalusi, saamislugu ning selle potentsiaali. Otseselt kriitilisi töid on avaldatud vähem, kuid nendest tooksin ma välja Berkeley ülikooli teadlaste poolt avaldatud artikli [4] pealkirjaga „Serverless Computing: Two Steps Forward, One Step Back“ (tõlge: „Serverivaba Töötlusmudel: Üks samm edasi, kaks sammu tagasi“). See artikkel keskendub eelkõige Amazoni Lambda teenusele ning toob kolme kasutusloo põhjal näiteid, miks FaaS lahendused on hetkeseisuga kõrge potentsiaaliga, kuid reaalsuses halvad valikud. Peamiste põhjustena tuuakse välja funktsioonide lühikesed eluead, sisendiga seotud pudelikaelad, aeglased liidestused ja puudulik riistvara. Samuti tuuakse töös välja võrdlused teiste pilveteenustega, kust tuleb välja, et nende valitud kasutuslugude jaoks on mõistlikum jätkata virtualiseerimise teenustega, sest see tuleks odavam ja kiirem. Töös tuuakse välja ka potentsiaalsed arendussuunad ja lahendused väljatoodud probleemidele. Nende hulgas on näiteks vajadusel andmete ja koodi ühes kohas hoidmine, riistvara valimine spetsiifilisteks kasutusjuhtumiteks ja ajapiiranguta instantsid.

Teine märkimist vääriv teadusartikkel [2] on kirjutatud samuti Berkeley ülikooli teadlaste poolt, kuid pakub ülevaate serverivaba töötluse saamisloost, selle potentsiaalsest suunast, hetkeprobleemidest, levinud müütidest ja nende alusest ning lõpuks annab ülevaate nende ennustustest. Lisaks sellele pakutakse erinevatele kasutuslugudele võrdlusi pilvefunktsioonide ja pilveserverite vahel. Artikli autorid võtsid hetkeprobleemide analüüsimisel aluseks selle, et kui kaugel on FaaS platvormide lahendused IaaS ja PaaS platvormide jõudlusest. Kõige ilmsem probleem oli see, et latentsust mõjutas enim suhtlus teiste teenustega nagu andmebaasid, kuid sellele probleemile pakuks leevendust *cache*'i kasutamine. Teiste probleemidena toodi välja raskused funktsioonide orkestreerimisel, et neid seotult kasutada, sest funktsioonid definitsiooni järgi ei oma olekut, mis tähendab, et muutujate väärtuste jagamine on raskendatud. Lisaks sellele mainiti ära ka suur latentsus funktsiooni esmakordsel käivitamisel ehk külmkäivitusel (ingl *cold start*), mis tulenes sellest, et esmalt on vaja eraldada funktsiooni jaoks ressursse, alla tõmmata rakenduse jaoks vastavad teegid ning alles siis rakendada äriloogikat. Töö autorid leiavad, et FaaS platvormide kasutajate arv tõuseb hüppeliselt, sest pilvefunktsioonide kasutamine on kõvasti lihtsam kui terve arhitektuuri ülesseadmine pilves. Lisaks nägid nad, et kui parandada latentsust teiste teenustega suhtlemisel, siis võib serverivaba arhitektuur olla võimalik lahendus palju rohkematele kasutuslugudele. Hinnastamise osas ennustasid töö autorid seda, et ajapikku on FaaS platvormide kasutamine veel odavam ning pikas

perspektiivis tähendab see seda, et see on odavam kui seniste laialt levinud „serveritega“ arhitektuuride kasutamine.

Olemasolevaid uurimustöid analüüsides on selge, et varasemalt on olnud FaaS platvormide võrdlused pigem ülevaatlikud ning ei ole piisavalt rõhku pandud platvormide omavahelisele sisulisele võrdlusele ja seetõttu keskendutakse selles töös just sellele aspektile. Lisaks sellele on varem avaldatud tööd olnud just pigem ülevaatlikud ning vähe on tehtud kasutuslugude põhjal võrdlusi.

3. Võrdlustestide ülesseadmine

FaaS platvormide võrdlus on üles ehitatud kahele kasutusloole, mida implementeeritakse Amazoni ja IBMi platvormidel. Valik osutus nende kasuks, sest Amazon [18] on suurim pilveteenuste pakkuja ning IBM OpenWhisk on vabavaral põhinev pilveplatvorm. Kasutuslugusid valides peeti silmas seda, et nad oleks fundamentaalselt erinevad ehk üks rakendus oleks õhuke funktsioon, mille ülesandeks on kirjeid andmebaasi kirjutada ning teine rakendus oleks arvutusressurssi nõudev. Kasutuslugude üles seadistamisel peeti silmas seda, et iga keskkonna server oleks samas piirkonnas teistega, et vältida regioonist tulenevaid latentsuse erisusi. Sellest tulenevalt valiti igal platvormil regiooniks London, sest see kattus igal platvormil.

Esimeseks kasutuslooks on lihtne veebirakendus, mis lubab kasutajatel registreeruda mingile kindlale üritusele. Kuna registreerimine avaneb kindlal kellaajal, siis võib sellel ajal olla registreeruda tahtjate arv märkimisväärselt suurem võrreldes mõne hilisema ajaga. See tähendab seda, et rakendus peab taluma mingitel kindlatel hetkedel suurt päringute arvu, kuid üldiselt rakendus väga palju ressursse ei vaja. Registreerumise salvestamiseks saadetakse HTTP päring, mille peab vastav funktsioon töötlema ja andmebaasi salvestama.

Teine kasutuslugu on rakendus, mis tegeleb pilditöötlusega, kus kasutaja laeb enda pildi üles ning seda töödeldakse rakenduse poolt ning tagastatakse seejärel kasutajale. Selline rakendus annab võimaluse hinnata serverivabade platvormide jõudlust sellistel ülesannetel, kus funktsiooni sisend on suhteliselt suure mahuga ning sellest tulenevalt funktsioon nõuab rohkem arvutusvõimsust.

Igal platvormil hinnatakse mõlemat kasutuslugu järgnevate kriteeriumite alusel: keskkonna üles seadmise lihtsus, latentsus külmkäivitamisel ja tavalise käivitamisel ajaperioodi jooksul, latentsus pikema aja jooksul, koormustest. Latentsus- ja koormustestid viiakse läbi Apache JMeter rakenduse abil, sest see on levinuim platvorm selliste testide tegemiseks.

Keskkonna üles seadmise lihtsuse hindamisel võetakse aluseks see, kui lihtne on esmakordselt ilma pilveplatvormide kogemust omamata rakendust üles seada, kui lihtne on probleemide tekkimisel leida abi dokumentatsioonist või teistest allikatest ning kui lihtne on siluda koodi vigade esinemisel.

FaaS platvormid üritavad optimeerida funktsioonide käivitamist ning kui mingit funktsiooni on vaja kindla perioodi jooksul mitu korda välja kutsuda, siis jäetakse see kindel funktsioon valmisolekusse. Sellest tulenevalt on järjekordne käivitamine (ingl *hot start*) oluliselt

kiirem, kui kutsuda välja funktsiooni, mida ei ole tükk aega välja kutsutud. Funktsiooni külmkäivitamine (ingl *cold start*) tähendab seda, et funktsiooni jaoks ei ole ühtegi aktiivset konteinerit, mis tähendab seda, et funktsiooni kasutamiseks on kõige pealt vaja üles seada konteiner ning alles siis on võimalik funktsioonile sisend anda ning vastav vastus tagastada. Aktiivse instantsi olemasolu sõltub funktsiooni viimasest välja kutsumise ajast ning seadistatud instantsi elueast. Viimane on alati ka piiratud platvormi enda poolt. Juhul, kui aktiivne instants on olemas, ehk tavalisel käivitusel, siis on funktsiooni väljakutse kiirem. Selle testi eesmärk on võrrelda funktsiooni külmkäivitamise ja nn sooja funktsiooni käivitamise latentsuse vahesid. Seoses sellega, et kuskil ei ole otseselt öeldud, kui pikk on iga funktsiooni instantsi eluiga, siis on tehtud mõned iseseisvad uuringud, kui pikk see AWS Lambda puhul võiks olla. Y. Cui poolt läbi viidud iseseisvate katsete [19] tulemus oli, et väiksemate mälueraldustega funktsioonide eluea ülemine piir oli ligi 62 minutit. Kuna IBM platvormi kohta ei ole sarnaseid uuringuid tehtud ega ei ole ka dokumenteeritud kuskil, kui pikk on keskmiselt funktsiooni instantsi eluiga, siis selle testi käigus eeldatakse, et see on sarnane AWS Lambda funktsiooni elueale. Testi jaoks luuakse esialgsele funktsioonile lisaks kaks koopiat, et testimist lihtsustada. Testi käigus saadetakse päringud kolmele külmale funktsioonile koopiale 5 minutiliste vahedega, kus iga kord saadetakse 120 sekundi jooksul 1000 päringut funktsioonile. Selleks, et veenduda, et funktsioon oleks külm luuakse need vahetult enne testimist.

Pikema aja jooksul latentsuse mõõtmise eesmärk on uurida seda, kas funktsiooni latentsuses tekivad mingid hälbeid, mis ei pruugi lühiajaliste testide puhul välja tulla ning analüüsida nende potentsiaalseid põhjuseid. Testi pikkuseks on 20 minutit ning on üles ehitatud nii, et külma funktsiooni pihta tehakse päringuid ja esimese 10 sekundi jooksul kogutakse kokku 100 kasutajat, kes teevad järgmise 20 minuti jooksul pidevalt päringuid.

Koormustesti eesmärk on testida iga platvormi automaatse skaleeruvuse toimimist juhul, kui päringute arv kasvab hüppeliselt ning päringute arv püsib suurena mingi ajaperioodi jooksul selleks, et uurida kui hästi platvormid saavad hakkama järsu päringute suurenemisega. Testi jooksul mõõdetakse keskmise päringu pikkust ning latentsuse ekstreemumeid. Keskmist päringu pikkust saab võrrelda eelmises punktis välja toodud üksikute käivitamistega. Testitakse külma funktsiooni instantsi ning esimese kolmekümne sekundi jooksul tuleb 500 kasutajat, kes üritavad registreeruda ning järgmise kolmekümne sekundi jooksul lisandub veel 1000 kasutajat sellele kasutajate hulgale. Pärast seda on kasutajate hulk stabiilne järgmised pool tundi ja selle ajavahemiku jooksul üritavad kõik

1500 kasutajat registreeruda. 30 minuti möödumisel kaovad 1000 kasutajat kolmekümne sekundi jooksul ära ning sellest järgmise kolmekümne sekundi jooksul kaovad 500 kasutajat ära. Pildi üleslaadimise kasutusloo puhul testitakse 500 kasutajaga 31 minuti jooksul, kus esimese ja viimase 30 sekundi jooksul vastavalt lisanduvad ja kaovad ära need kasutajad.

Esimese kasutusloo põhjal valiti funktsiooni parameetriteks mäluhulk 256MB ja maksimaalne käitusaeg 30s ning teise kasutusloo jaoks sai valitud vastavalt 1024MB ja 60s. Ühtlasi teostatakse kõiki teste samades tingimustes ehk iga testi käitusajaks on testarvutis välja lülitatud kõik muud protsessid ning üles- ja allalaadimiskiirus on internetil 20Mbit/s.

3.1.Esimene kasutuslugu – veebivormi salvestamine

Kasutusloo stsenaarium on selline, et mingile üritusele registreerumine avaneb kindlal kellaajal ning kuna tegu on populaarse üritusega, siis on sellel kellaajal registreeruda soovijate arv väga suur, mis tõttu peab registreeringute salvestamine suutma toime tulla äkilise kasutajate arvuga. Sellele kasutusloole sarnaseid stsenaariume, kus päringute arv kasvab hüppelist on palju – e-poe soodusmüük algab kindlal kellaajal; veebileht saab kajastust ning saab üleöö kuulsaks, mis tõttu külastajate arv kasvab. Registreerumise teostamiseks on vaja kasutajalt tema eesnime, perekonnanime ja emaili ning seejärel on vaja saata päring koos eelneva infoga funktsioonile..

3.1.1. Registreerumise vormi loomine

Registreerumise vorm koosneb lahtritest eesnimi, perekonnanimi ja email. Selle jaoks lõin illustratiivse HTML vormi, mida on näha Joonisel 5. Vormil „registreerun“ nuppu vajutades saadetakse päring vastavale funktsioonile. Päringu sisuks on `application/x-www-form-urlencoded` kujul eesnimi, perekonnanimi ja email. Näiteks on ühe näitepäringu kuju selline: `first_name=Mark-Eerik&last_name=Kodar&email=mark.kodar@gmail.com`

Joonis 5. Registreerimisvorm.

3.1.2. IBM OpenWhisk registreerumise funktsiooni üles seadmine

Funktsiooni üles seadmiseks on vajalik luua eelnevalt IBM keskkonnas kasutaja. Pärast kasutaja loomist saab luua vastava *actioni*, mis ongi funktsioon, mis hakkab sisendit töötlema. Funktsiooni loomisel on vaja valida sellele nimi, soovi korral ka grupp, kuhu funktsioon kuuluda võiks, ning sobiv käitusmootor ehk, mis programmeerimiskeeles funktsioon kirjutatakse.

Pärast funktsiooni loomist on vajalik luua IBM Cloud platvormil andmebaasiteenus, milleks on NoSQL andmebaas Cloudant, mis hoiab JSON dokumente. NoSQL andmebaas [20] on lihtsustatult mitterelatsiooniline andmebaas, mis võib olla XML, graafide, dokumentide või objektide andmebaas. Enamik NoSQL andmebaase on lihtsad võtme ja väärtuse paaride hulgad. Teenuse loomisel on vajalik valida teenuse nimi, geograafiline regioon, kuhu juurutada teenus, märgend ning autentimismeetodid. Lisaks sellele on vajalik valida hinnastus. Regiooni valikul on mõistlik silmas pidada seda, et mida lähemal see regioon on kasutajatele, seda väiksem on latentsus päringutel.

Pärast seda on vaja luua pääsumandaadid (ingl *credentials*), mille abil genereeritakse andmebaasi teenusesse sisenemiseks vajalikud API võtmed, kasutajanimi, parool ja URL. Neid mandaate on vaja selleks, et hiljem nende abil funktsioonis kasutada andmebaasi teenust.

Järgmise sammuna saab lisada funktsioonile parameetrid ehk väärtused, mis tulevad lisaks päringust tulevatele parameetritele kaasa. Nende parameetrite hulka tuleb lisada eelpool loodud mandaadi väärtused. Selleks, et neid väärtuseid koodis kasutada, tuleb need põhifunktsioonile antud väärtuste sõnastikust välja võtta.

Sarnaselt mandaadi väärtustele saab funktsioonile päringuga saadud info kätte funktsiooni main parameetrist, mis on sõnastik. Nendest väärtustest tuleb luua omakorda salvestuseks vajalik sõnastik, mida salvestada siis hiljem andmebaasi. Joonisel 6 on näha, et esialgu võetakse põhifunktsiooni main parameetrist välja ees- ja perekonnanimi ning email ja seejärel luuakse neist registreeringu objekt, mis lisatakse andmebaasi funktsioonis add_registration. Seal funktsioonis luuakse API-võtme ja kasutajanimega andmebaasiteenusega ühendus ning luuakse hiljem dokument, mis tagastatakse päringu lõpus.

```
10  import sys
11  import json
12  from cloudant.client import Cloudant
13
14  def add_registration(reg, username, apikey):
15      database_name = "registration"
16      client = Cloudant.iam(username, apikey, connect=True)
17      database = client[database_name]
18      return database.create_document(reg)
19
20
21  def main(param):
22      first_name = param['first_name']
23      last_name = param['last_name']
24      email = param['email']
25      registration = {'first_name': first_name, 'last_name': last_name, 'email': email}
26      new_doc = add_registration(registration, param['username'], param['apikey'])
27      return new_doc
```

Joonis 6. Registreeringuid haldava funktsiooni kuvatõmmis IBM OpenWhiskis.

3.1.3. AWS Lambda registreerumise funktsiooni ülesseadmine

Sarnaselt IBM OpenWhiskile on ka AWS Lambda kasutamiseks vajalik eelnev registreerimine Amazoni keskkonnas. Pärast kasutaja loomist on võimalik luua AWS Lambda funktsioon. Funktsiooni loomiseks on vaja eelnevalt valida regioon ning pärast seda on võimalik vastavas regioonis luua teenuseid. Seejärel on võimalik valida funktsioonile nimi ning käitusmootor ja kasutajaõiguste grupp.

Pärast funktsiooni loomist tuleb luua DynamoDB tabel, kuhu vajalikud andmed sisestada. DynamoDB on [21] võtmete ning väärtuste paaride ja dokumentide NoSQL andmebaas. Tabeli loomisel on vajalik valida võtmeväärtus, milleks selle kasutusloo puhul on email. Pärast seda tuleb luua eraldi kanne tabelisse, et saaks tekitada väljad eesnime ja perekonnanime jaoks.

Pärast tabeli loomist tuleb valida Lambda konsooli Designer vaates teised Amazoni teenused, millega funktsioon suhtlema hakkab. Antud kasutusloo jaoks on vajalik API Gateway ja eelpool mainitud DynamoDB. DynamoDB puhul tuleb veenduda, et seadistuses on loodud vastav IAM roll, et funktsioonil oleks andmebaasi kirjutamise ja lugemise õigus. API Gateway on [22] platvorm, mis lubab kasutajatel luua, hallata, turvata ja monitoorida erinevaid Amazoni teenustega seotud REST *endpointe*. Selle abil on võimalik luua ka väliseid *endpointe*, et välja kutsuda erinevaid tegevusi Amazoni teenustes. HTTP meetoditest on toetatud GET, POST, PUT, PATCH ja DELETE. API Gateway seadistuste all on vaja luua registreerumise vormi salvestamiseks POST endpoint ning muuta ära vaikimisi seade, mis üritab parsida ainult JSON kujul tulevaid dokumente, kuid eelnevalt

loodud vorm saadab `application/x-www-form-urlencoded` kujul päringu. Pärast seda tuleb API avaldada, et seda saaks päringutel kasutada.

AWS Lambda põhifunktsiooniks on `lambda_handler`, mis võtab parameetriks sündmuse ning konteksti. Sündmuses on kirjas vastavalt selle andmed ning on sõnastiku kujul, kuid seda peab vajadusel teisendama sobivale kujule olenevalt päringu määratud andmetüübist ja selle teisendamisest. Kontekstis info funktsiooni käituse kohta, milleks on näiteks funktsiooni nimi, versioon ja eraldatud mälu ning veel detailset infot.

Selle kasutusloo tarvis on vaja teisendada sündmust, et saada kõik vajalikud parameetrid õigele kujule. Seda teeb funktsioon `parseToDict`, mis parsib sündmuse sisu sõnastiku kujule. Pärast seda on võimalik Boto3 abil salvestada andmed DynamoDB andmebaasi. Boto3 on Amazoni enda SDK Python programmeerimiskeele jaoks, mille abil on võimalik ligi pääseda teistele Amazoni teenustele. Joonisel 7 on näha funktsiooni tema valmiskujul.

```
9  # https://edtheron.me/projects/store-messages-aws-dynamodb-lambda-api-gateway-cognito
10 def respond(err, response=None):
11     return {
12         'statusCode': '400' if err else '200',
13         'body': err if err else json.dumps(response),
14         'headers': {
15             'Content-Type': 'application/json',
16         },
17     }
18
19 def parseToDict(event):
20     # necessary for parsing @ sign
21     urlencoded = urllib.parse.unquote(event['body'])
22     # converts x-www-form-urlencoded to dict
23     return {x.split('=')[0]: x.split('=')[1] for x in urlencoded.split("&")}
24
25 def lambda_handler(event, context):
26     event_json = parseToDict(event)
27     first_name = event_json['first_name']
28     last_name = event_json['last_name']
29     email = event_json['email']
30     table = boto3.resource('dynamodb').Table('registration')
31     data = {
32         'first_name': first_name,
33         'last_name': last_name,
34         'email': email
35     }
36     table.put_item(Item=data)
37     return respond(None, data)
```

Joonis 7. Registreerimist haldav funktsiooni kuvatõmmis AWS Lambda platvormil.

Lisaks sellele on vaja ära defineerida funktsioonile eraldatav mälu ning tema käituse maksimaalne ajahulk. Selle kasutusloo jaoks sai valitud mäluhulgaks 256MB ja maksimaalseks käitusajaks 30 sekundit.

3.1.4. Jõudlustestide seadistamine Apache JMeter rakenduses

Selle kasutusloo testimiseks on vaja luua testskriptid, mis saadaks päringu funktsioonile pilves koos infoga, mis on registreerimiseks vajalik. Selleks, et igas päringus oleks unikaalsed väärtused, kasutan ma rakendusse sisse ehitatud UUID väärtuse loojat, mis tagab selle, et ükski päring ei sisalda sama infot. Seega saavad väljade nimeks genereeritud järgmise mustriga: `UUID_first, UUID_last, UUID@test.ee`.

Järgmiseks on vaja valida HTTP meetod ja andmetüüp. milleks on selle kasutusloo puhul POST ning andmetüübiks on `application/x-www-form-urlencoded`. Meetodi sihtaadress on vastavalt funktsioonile tema enda URL. Pärast seda on vaja seadistada lõimede ehk arv vastavalt sellele, mis liiki test on.

3.2.Teine kasutuslugu – pilditöötlus

Teiseks kasutuslooks valiti pilditöötluseks selle pärast, et kuna reeglina võtab pilditöötlus palju arvutusressurssi, siis selle ressursi pidev töös hoidmine läheb väga kulukaks. FaaS platvormid pakuvad sellele alternatiivi sellel moel, et vastavaid ressursse luuakse ainult juhul, kui neid reaalselt vaja on. Samuti on see kasutuslugu erinev esimesest kasutusloos selles mõttes, et andmebaasiga suhtlust ei toimu ning saaks uurida puhtalt funktsiooni enda jõudlust.

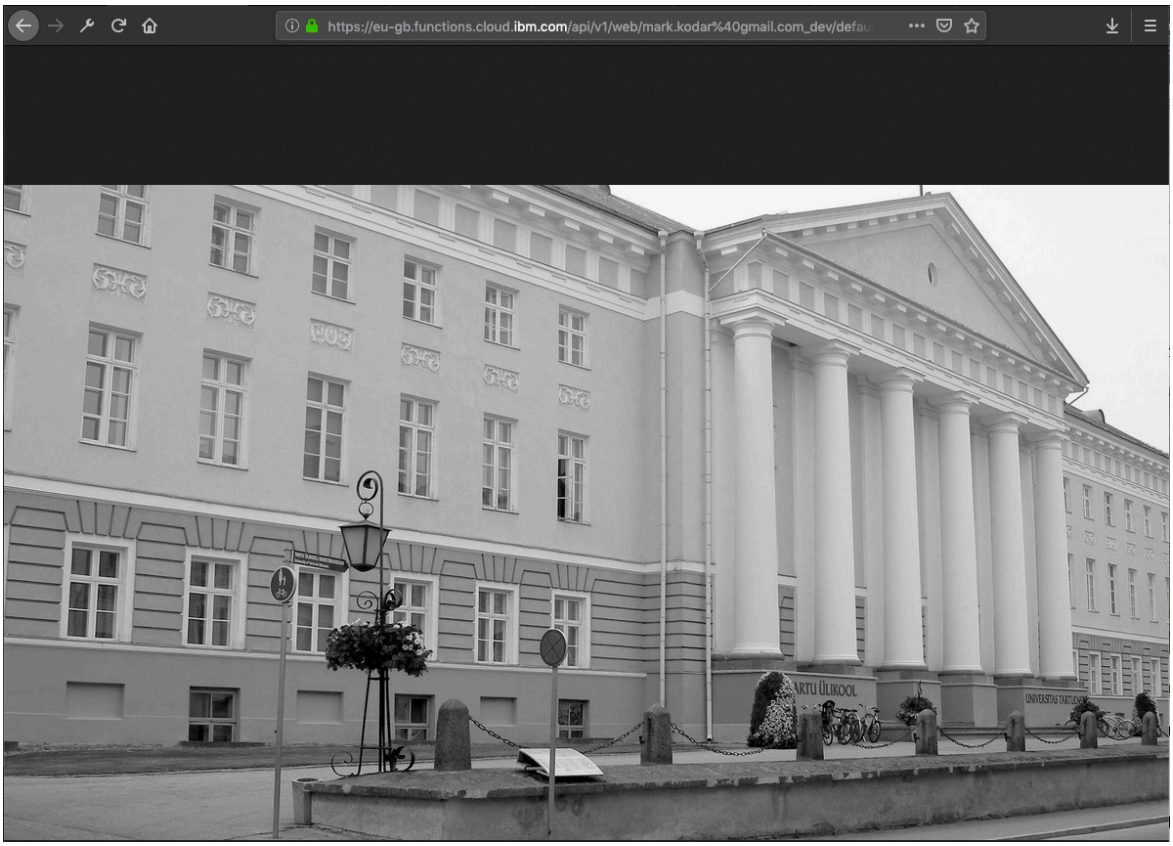
Teise kasutusloo puhul peeti silmas seda, et kuna pildi- ja videotöötluseks on vaja tavaliselt mooduleid, mida standardteekide hulgas ei ole, siis see kasutuslugu ilmestaks ka seda, kui lihtne või raske on lisada funktsioonidele mooduleid, mis ei ole standardteegis. Pilditöötluse jaoks sai valitud OpenCV2 moodul, mis töötleb kasutaja poolt üleslaetud pilti must-valgeks ning seejärel tagastab selle. Pilditöötluse ajal faili andmebaasi salvestamist ei toimu, vaid faili töödeldakse jooksvalt funktsiooni sees.

3.2.1. Pildi üleslaadimise vormi loomine

Pildi üleslaadimiseks on loodud lihtne HTML vorm, kus kasutaja saab enda arvuti failisüsteemist valida faili, mille ta soovib muuta must-valgeks ning seejärel vajutab nupule. Nupp saadab `multipart/form-data` kujul selle info vastavale FaaS platvormile, mis tegeleb selle töötlusega ning tagastab selle. Joonisel 8 on näha pildi üleslaadimis vormi ning joonisel 9 on näha pilti originaalkujul ning joonisel 10 on näha pilt pärast töötlust.

Joonis 8. Pildi üles laadimise vorm.

Joonis 9. Töödeldav pilt originaalkujul. [10]



Joonis 10. Pilt pärast töötlemist veebilehitseja aknas.

3.2.2. IBM OpenWhisk funktsiooni loomine

Selleks, et IBM OpenWhisk platvormil kasutada teeke, mida ei ole vaikimisi toetatud on vaja seda teha läbi Dockeri tõmmiste. Dockeri tõmmis peaks põhinema IBMi enda vastava

[10] https://www.visitestonia.com/en/main-building-of-the-university-of-tartu

käitusmootori tõmmisele ning sellele tuleb lisada vastava käskude abil teegid, mida soovitakse kasutada. Seejärel on vaja vastav konteiner ehitada ning lisada see Docker Hub keskkonda. Pärast seda tuleb arhiveerida vajalikud koodifailid ning läbi käsurea luua uus funktsioon. Käesoleva funktsiooni loomiseks oli selleks vaja sisetada järgmine käsk:

```
ibmcloud fn action create process_image func.zip --docker markkod/openwhisk_opencv:1.1
```

Pärast seda käsku on funktsiooni parameetreid hallata ka veebikeskkonnas, kuid sellisel kujul funktsioonide loomine tähendab seda, et selle lähtekoodi ei saa graafilises kasutajaliideses muuta ning koodi muutmiseks peab seda tegema lokaalselt ning uuesti käsurealt failid üles laadima. Joonisel 11 on näha funktsiooni, mis laeti üles platvormile.

```
def main(param):
    png = param["__ow_body"]
    bindata = base64.b64decode(png)
    fp2 = io.BytesIO(bindata)
    form = cgi.FieldStorage(fp=fp2, environ={'REQUEST_METHOD': 'POST',
                                             'CONTENT_LENGTH': len(bindata),
                                             'CONTENT_TYPE': param["__ow_headers"]["content-type"]})
    dat = form["pic"]
    img_bytes = dat.file.read()
    np_arr = np.frombuffer(img_bytes, dtype='uint8')
    img_np = cv2.imdecode(np_arr, cv2.IMREAD_UNCHANGED)
    gray = cv2.cvtColor(img_np, cv2.COLOR_BGR2GRAY)
    retval, buffer = cv2.imencode('.jpg', gray)
    out = base64.b64encode(buffer)
    return {
        "isBase64Encoded": True,
        "statusCode": 200,
        "headers": {"content-type": "image/jpeg"},
        "body": out
    }
```

Joonis 11. Funktsioon IBM OpenWhisk platvormil.

Funktsioonis esiteks loetakse välja tema parameetritest `multipart/form-data` kujul saadetud pilt, mis ära kodeeritakse ning hiljem loetakse välja päringu päistest välja päringu andmetüüp. Pärast seda loetakse vormist baidid, millest luuakse järjend. Sellest luuakse OpenCV2 abil pilt ning töödeldakse see mustvalgeks. Pärast seda teisendatakse pilt jälle baitideks ning kodeeritakse ära enne tagasi saatmist.

3.2.3. AWS Lambda funktsiooni loomine

AWS platvormil tuleb sarnaselt eelmisele kasutusloole üles seada funktsioon, mille käitusmootor on Python 3.6. Kuid selle kasutusloo juures tuleb eraldi Dockeri tõmmise abil OpenCV2 teek üles laadida. Selle jaoks kasutatakse ühte Dockeri tõmmist, kus on see vastav moodul olemas. Sarnaselt IBM OpenWhisk platvormile on ka selle Dockeri tõmmise aluseks platvormihaldaja poolt loodud tõmmis.

Pärast funktsiooni ülesseadmist on vaja lisada funktsioonile juurde kiht (ingl *layer*), kuhu tuleb lisada eelpool mainitud tõmmis kokku pakituna ning salvestada see uue kihina. Pärast kihi lisamist on võimalik neid mooduleid kasutada Lambda funktsioonis. Selleks, et funktsiooni saaks vajalikul moel välja kutsuda, siis on vaja luua ka API Gateway abil vastav endpoint ning seadistada nii, et see oleks võimeline vastu võtma binaarset meediatüüpi ehk suudaks üles laetud failidega toime tulla ning suudaks neid hiljem ka tagastada sellisel kujul. Funktsioon on identne sellega, mis loodi IBM OpenWhisk platvormi jaoks ning seda on võimalik näha joonisel 11.

3.2.4. Jõudlustestide loomine Apache JMeter rakenduses

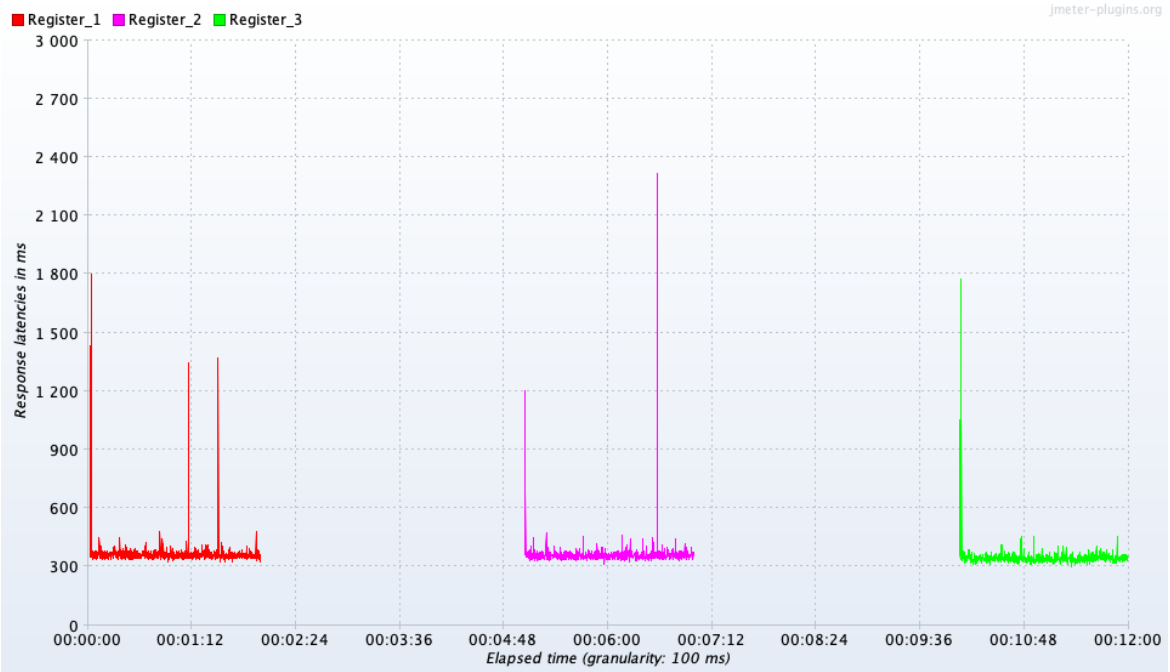
Selle kasutusloo jaoks on vaja laadida üles pilt läbi vormi, mis tähendab, et kasutada `multipart/form-data` andmetüüpi, et pildiinfo saata funktsioonile. Lisaks sellele on vaja ära defineerida, millist pilti üles laetakse ning määrata arvutis selle asukoht, pildi üleslaadimis HTML vormi välja nimi ning MIME tüüp. Antud testi jaoks kasutatakse sama pilti iga päringu jaoks ning seda pilti on võimalik näha joonisel 9. Pärast seda on vaja ära määrata funktsiooni sihtaadress ning seadistada lõimed vastavalt testile.

4. Tulemused ja analüüs

Selles peatükis antakse ülevaade tehtud testide tulemustest mõlemal platvormil, kus iga testi kohta on eraldi alampeatükk, milles uuritakse latentsute erinevusi ja päringute õnnestumisi igal platvormil ja kasutuslool ning viimase alampeatükina on tulemuste analüüs.

4.1.Külmkäivituse testi tulemused

Joonisel 12 on näha AWS Lambda platvormil tehtud külmkäivituse testi latentsuste graafikut, kus loodi kolm identset funktsiooni, et testida nende külmkäivitust efektiivsemalt. Graafikult on selge, et külma funktsiooni esmakordselt välja kutsudes on latentsus mitu korda suurem võrreldes keskmise latentsusega, kuid selle testi puhul külmkäivitus väga pikka aega mõju ei avaldanud latentsustele. Joonisel 13 olevalt kokkuvõttest on näha, et iga funktsiooni koopia pakkus ligilähedaselt sama keskmist latentsust, mis jäi 370 ja 380 ms vahele. Teisalt erinesid nende standardhälbed ning see kajastub ka joonisel 12, kus on näha, et esimese ja teise funktsiooni koopia puhul tekkis pärast külmstarti ekstreemumeid.

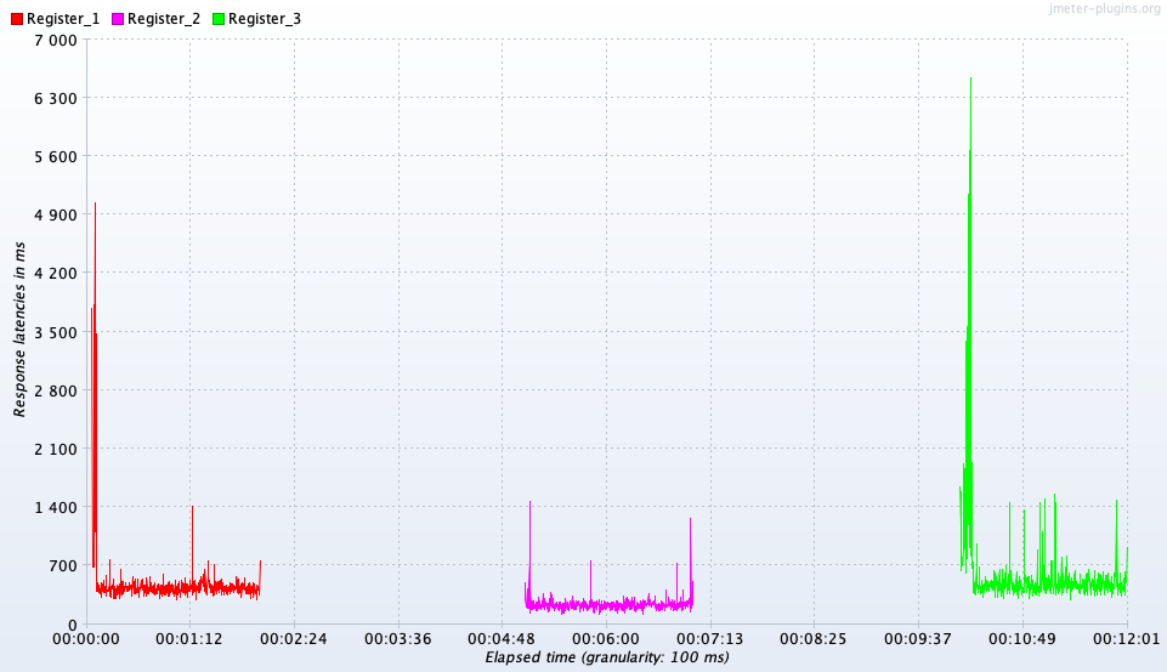


Joonis 12. Registreerimise kasutusloo külmkäivituse testi latentsuste graafik AWS Lambda platvormil.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Register_1	1000	380	315	2348	148.34	0.00%	8.3/sec	4.56	3.20	561.0
Register_2	1000	378	316	2321	124.47	0.00%	8.4/sec	4.61	3.24	561.0
Register_3	1000	374	290	2098	181.11	0.00%	8.5/sec	4.65	3.26	561.0
TOTAL	3000	377	290	2348	153.10	0.00%	4.2/sec	2.28	1.60	561.0

Joonis 13. Registreerimise kasutusloo külmkäivituse testi kokkuvõte AWS Lambda platvormil.

Joonisel 14 on näha sama testi tulemused IBM OpenWhisk platvormil ning seal loodi samuti kolm koopiat ühest funktsioonist. Selle testi tulemused näitavad seda, et ka seal on külmkäivitus ilmselge, kuid kahel juhul kolmest see hiljem mõju ei avaldanud. Kuid ühel juhul on näha, et Register_3 puhul oli külmstardi probleem kõige tõsisem ning see avaldas mõju ka hilisematele päringutele, mida AWS Lambda puhul väga näha ei olnud. Samuti on näha joonisel 15, et kolme funktsiooni koopia keskmised latentsused erinesid omavahel mitmekordselt. Madalaim latentsus oli 246 ms, keskmine 588 ms, suurim 761 ms ning see andis üleüldiseks keskmiseks 532 ms, mis on kõrgem kui AWS Lambda keskmine latentsus.



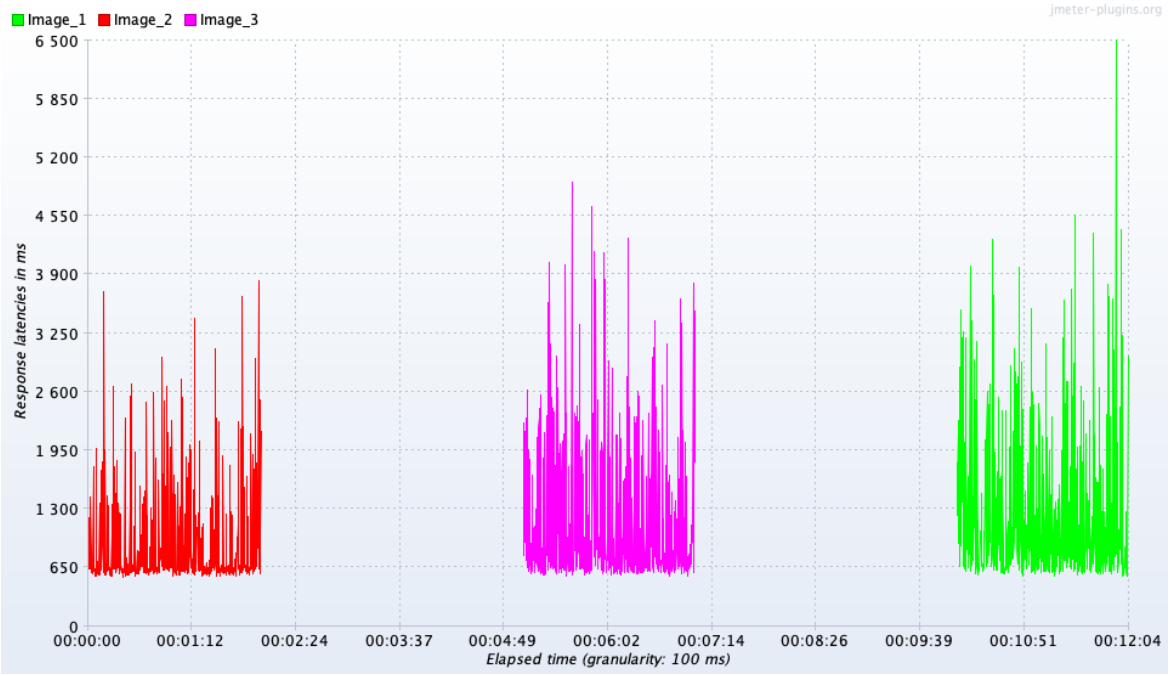
Joonis 14. Registreerimise kasutusloo külmkäivituse testi latentsuste graafik IBM OpenWhisk platvormil.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Register_1	1000	588	292	6249	761.73	0.00%	8.3/sec	8.65	3.39	1067.0
Register_2	1000	246	116	2327	133.26	0.00%	8.6/sec	6.85	3.49	820.0
Register_3	1000	761	295	6549	1031.69	0.00%	8.5/sec	8.88	3.48	1067.0
TOTAL	3000	532	116	6549	774.56	0.00%	4.2/sec	4.00	1.70	984.7

Joonis 15. Registreerimise kasutusloo külmkäivituse testi kokkuvõte IBM OpenWhisk platvormil.

Teise kasutusloo puhul AWS Lambda platvormil on näha joonisel 16, et kui funktsioon peab tegema arvutusmahukamat ülesannet siis on funktsiooni latentsus ebastabiilne, sest latentsused kõikusid üsna palju pärast esialgset väljakutsed ning seda on ka näha joonisel 17, kus iga koopia standardhälve on küllaltki suur ning üleüldine keskmine standardhälve

tuli 803.78 ms. Üleüldine keskmine latentsus tuli 1120 ms, mis on ligi kolm korda suurem, kui esimese kasutusloo puhul.



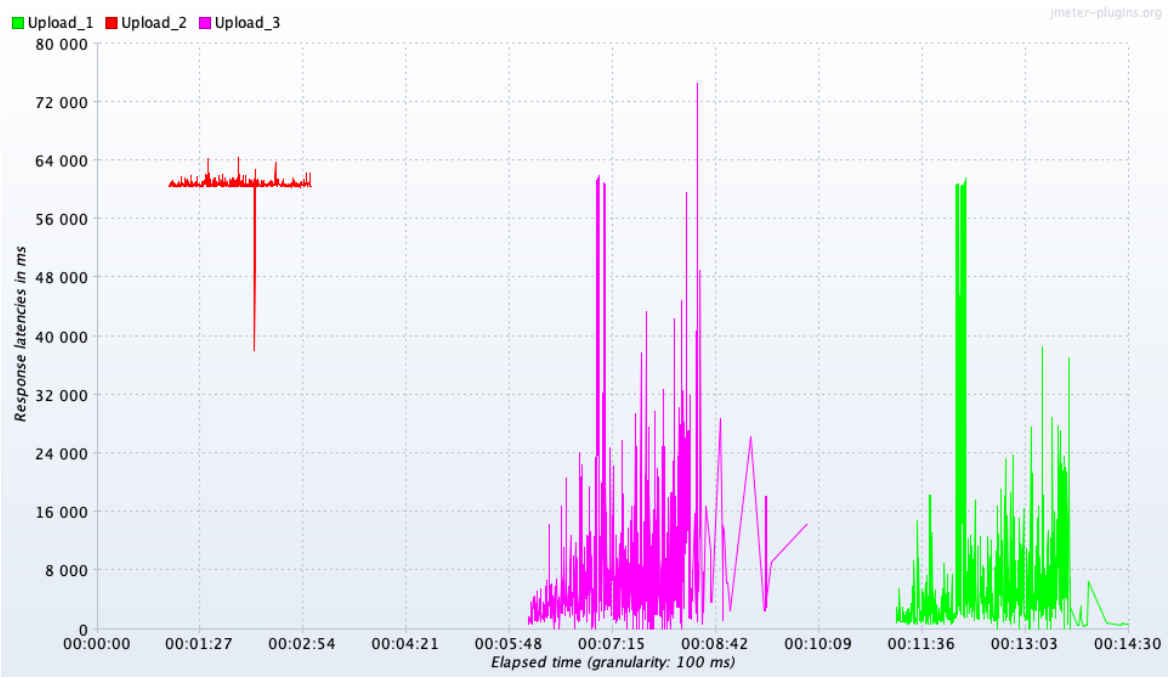
Joonis 16. Külmkäivituse testi latentsused AWS Lambda platvormil pildi üleslaadimise kasutuslool.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Image_2	1000	894	545	4230	593.54	0.00%	8.2/sec	2.78	1589.54	346.0
Image_3	1000	1177	550	5936	822.28	0.00%	8.2/sec	5.98	1586.91	745.0
Image_1	1000	1288	557	6509	909.45	0.00%	8.3/sec	6.04	1603.72	745.0
TOTAL	3000	1120	545	6509	803.78	0.00%	4.1/sec	2.48	799.89	612.0

Joonis 17. Pildi üleslaadimise kasutusloo külmkäivituse testi kokkuvõte AWS Lambda platvormil.

Joonisel 18 on näha sama testi latentsuse graafik IBM OpenWhisk platvormil ning see graafik näitab seda, et suuremate koormuste puhul on see platvorm külmkäivitustel palju vähem etteaimatavam, sest funktsiooni esimene koopia tegi märkimisväärselt halvema tulemuse kui teised kaks koopiat. Selle üheks põhjuseks võib olla see, et platvorm nö valmistas teised konteinerid ette, sest esimene koopia (Upload_2) sai niivõrd palju koormust. Ühtlasi on näha joonisel 19 olevast tabelist, et esimene funktsiooni koopia saatis tagasi keskmiselt väga väiksed päringuvastused, mis ei sisaldanud endas korrektselt töödeldud pilti. Samuti on näha tabelist, et esimese koopia keskmine latentsus oli vastavalt ligi kolm ja kuus korda suurem kui teise (Upload_3) ja kolmanda (Upload_1) keskmine latentsus. Teisalt oli esimese koopia päringute veamäär märkimisväärselt väiksem 0.2%

juures, kui teise koopia 8.45% ja kolmanda koopia 2.73%. Selle esimese koopia tulemus on tõenäoliselt erind ning väga palju ei saa selle tulemusest välja lugeda, kuid see test näitab seda, et funktsiooni registreerimise järel võib väljakutse olla kümme korda aeglasem tavakasutusest.



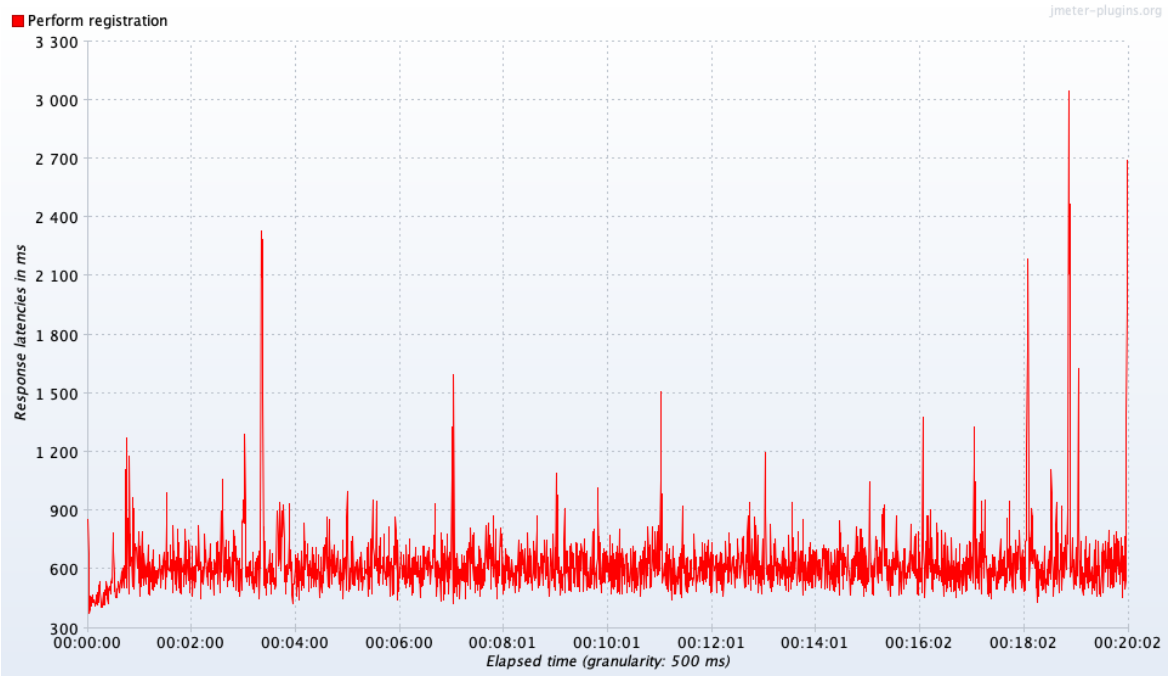
Joonis 18. Külmkäivituse testi latentsused IBM OpenWhisk platvormil pildi üleslaadimise kasutuslool.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Upload_2	1000	60750	15558	64608	2090.30	0.20%	5.5/sec	4.75	1070.09	877.0
Upload_3	959	22000	1006	190215	22258.66	8.45%	4.0/sec	1198.20	723.21	305653.0
Upload_1	952	10807	694	61678	12374.28	2.73%	4.6/sec	1452.02	873.29	321771.1
TOTAL	2911	31651	694	190215	26047.06	3.74%	3.3/sec	673.41	626.86	206226.2

Joonis 19. Pildi üleslaadimise kasutusloo külmkäivituse testi kokkuvõte IBM OpenWhisk platvormil.

4.2.Pikaajalise testi tulemused

Joonisel 20 on näha registreerimispäringute latentsust 20 minuti jooksul AWS Lambda platvormil. Sealt jooniselt on ühtlasi näha, et üldiselt on latentsus stabiilselt 600 ms ümbruses, kuid mõningatel hetkedel on näha hälbeid, mis esinevad suhteliselt harva, kuid testi lõpupoole esineb neid rohkem. Kõige tugevamate hälvete põhjusteks oli tõenäoliselt funktsiooni käitusaja ületamine, mis oli 30 sekundit ning seda oli ka näha päringute latentsuste tabelist. Need päringud olid ka ainukesed, mis said vastuseks veateate. Üldiselt oli latentsuste ülemine piir 3000 ja 4000 ms vahel. Testi kokkuvõttest joonisel 21 on näha, et keskmine latentsus oli 613 ms ja standardhälve 296.78 ms.



Joonis 20. Pikaajalise test latentsused AWS platvormil registreerimise kasutuslooga.

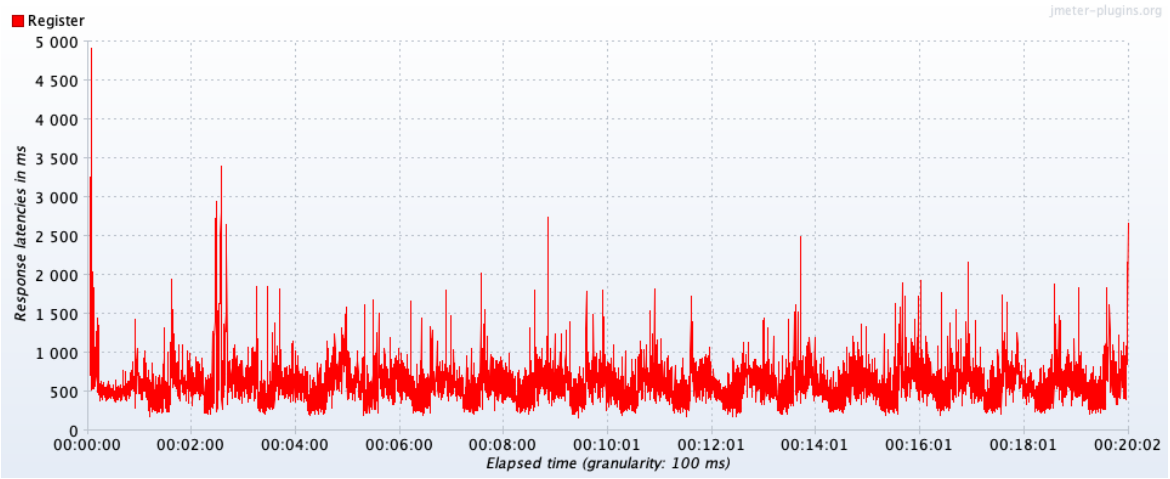
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Perform regi...	193822	613	302	29468	296.78	0.00%	161.2/sec	88.33	62.66	561.0
TOTAL	193822	613	302	29468	296.78	0.00%	161.2/sec	88.33	62.66	561.0

Joonis 21. Pikaajalise testi kokkuvõte AWS platvormil registreerimise kasutuslooga.

IBM OpenWhisk platvormil pikaajalist registreerimistesti kokkuvõtvast tabelist joonisel 22 on näha, et sellist koormust ei suuda platvorm vastu võtta, sest veateate saanud päringute osakaal oli 77.77%. Selle põhjuseks oli andmebaasi kirjutamise ja lugemise piirangud, mis olid vaikimisi seatud. Kusjuures Amazoni platvormil sellega probleeme ei tekkinud, et andmebaas oleks piirama hakanud päringute arvu. Kuna paljud päringud said veateate, siis sellest tulenevalt oli ka keskmine latentsus madalam ehk 558 ms ning standardhälve oli 378.02 ms. Seda on võimalik ka näha joonisel 23, kust on näha, et äärmuslikud väärtused tekkisid ainult pikaajalise testi alguses ning selle põhjuseks oli tõenäoliselt külm funktsioon.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Register	212930	558	108	8008	378.02	77.77%	177.1/sec	144.87	72.30	837.6
TOTAL	212930	558	108	8008	378.02	77.77%	177.1/sec	144.87	72.30	837.6

Joonis 22. Pikaajalise testi kokkuvõte registreerimise kasutuslooga IBM OpenWhisk platvormil.

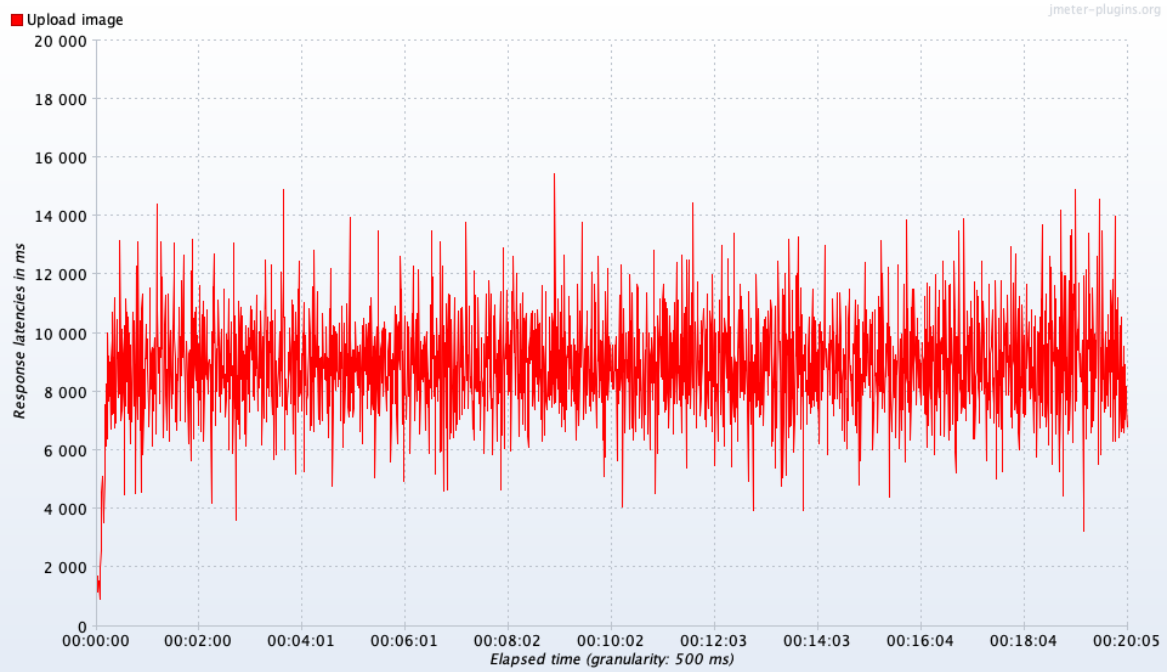


Joonis 23. Pikaajalise test latentsuste graafik registreerumise kasutuslooga IBM OpenWhisk platvormil.

AWS Lambda platvormil pildi üleslaadimise latentsuste graafikust joonisel 25 on näha, et latentsus oli palju suurem võrreldes esimese kasutuslooga, kuid see tulenes sellest, et teine pilditöötlus nõuab rohkem ressurssi võrreldes lihtsa andmebaasi salvestamisega. Jooniselt 24 on näha, et keskmiseks latentsuseks tuli 8930 ms ehk ligi 9 sekundit ja standardhälbeks oli 3510 ms. Selline latentsus on liiga suur aeg veel lihtsa pilditöötlusülesande korral ning eriti arvestades, et tegu ei olnud isegi koormustestiga. Selle kasutusloo testi latentsuste graafikust on näha, et latentsused püsisid stabiilsemana ning oli vähem äärmuslike väärtusi võrreldes esimese kasutuslooga. Teise kasutusloo puhul ei tulnud mitte ühtegi veateadet.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Upload image	13411	8930	678	54915	3510.62	0.00%	11.1/sec	5.24	2146.75	482.0
TOTAL	13411	8930	678	54915	3510.62	0.00%	11.1/sec	5.24	2146.75	482.0

Joonis 24. Pilditöötluse kasutusloo pikaajalise testi kokkuvõte AWS Lambda platvormil.

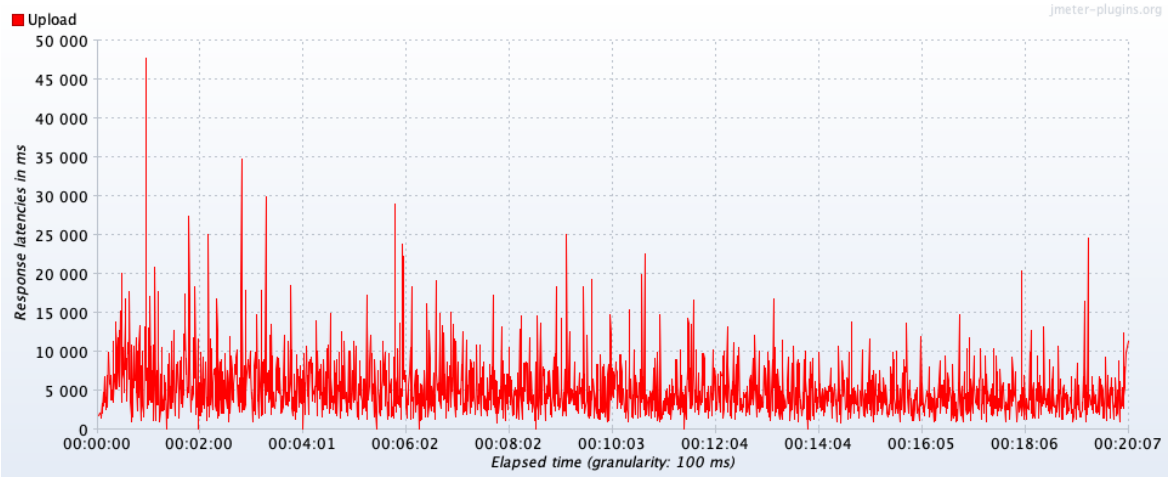


Joonis 25. Pilditöötluse kasutusloo latentsuste graafik pikaajalise testi korral AWS Lambda platvormil.

Pildi üleslaadimise kasutuslool oli IBM OpenWhisk platvormi tulemused paremad võrreldes samal platvormil registreeringute tegemisega, sest selle kasutusloo käigus ei pöördutud kordagi andmebaasi poole. Sellest tulenevalt on joonisel 26 näha, et veamäär oli palju väiksem ehk 1.25%, kuid seda ei anna võrrelda AWS Lambda veamääraga samades tingimustes, milleks oli 0%. Samuti on võrreldes teise platvormiga näha, et sellel platvormil tekkis rohkem äärmuslikke latentsusi, mille põhjustajaks olid funktsiooni käitusaja ületamised.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Upload	7674	9693	1104	450895	16019.09	1.25%	6.4/sec	2099.60	1213.12	338291.1
TOTAL	7674	9693	1104	450895	16019.09	1.25%	6.4/sec	2099.60	1213.12	338291.1

Joonis 26. Pilditöötluse kasutusloo pikaajalise test kokkuvõte IBM OpenWhisk platvormil.



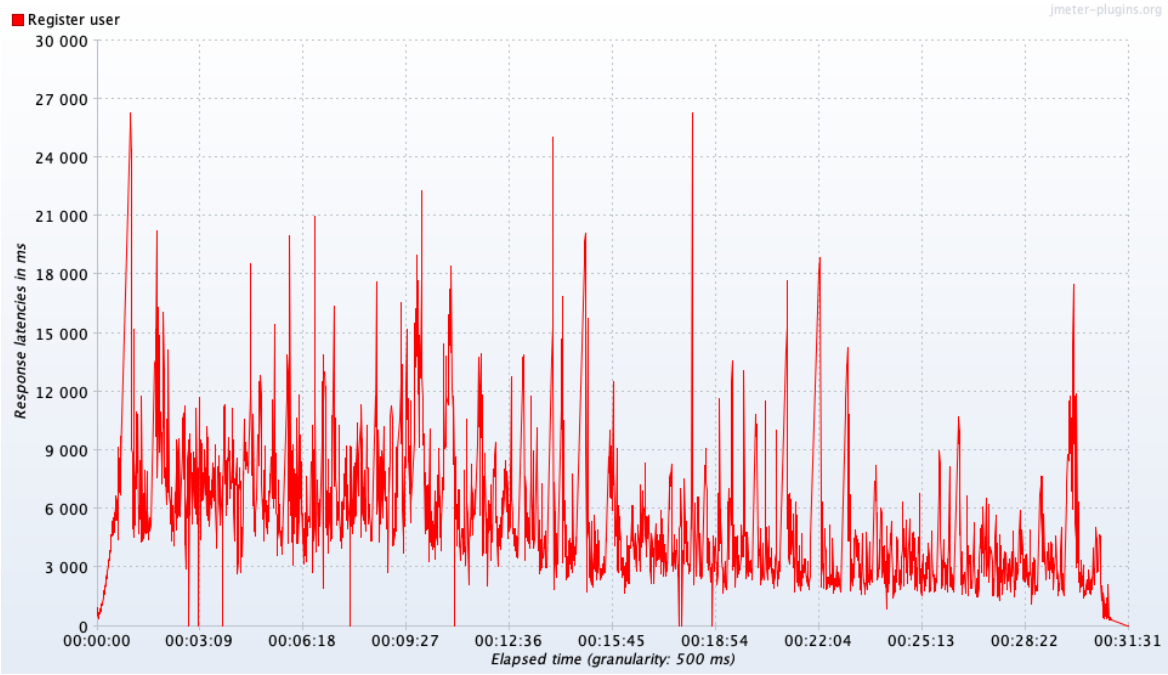
Joonis 27. Pilditöötluse kasutusloo latentsuste graafik pikaajalise testi korral IBM OpenWhisk platvormil.

4.3.Koormustesti tulemused

Registreerimise kasutusloo AWS Lambda platvormi koormustestis tuli välja, et külmal funktsioonil on väga raske toime tulla suurte päringute arvuga. Jooniselt 29 on näha, et esimesed 30 sekundit oli päringute madalama hulga tõttu keskmine latentsus madalam. Pärast 30 sekundit päringute arv kasvas järgmise 30 sekundi jooksul ning funktsioonil oli esialgu väga raske toime tulla selliste päringute arvuga. Esimesed kümme minutit püsis latentsus suhteliselt kõrgel ning pärast seda toimus aeglane latentsuse stabiliseerumine, kuid ikkagi jäi keskmine väärtus väga kõrgeks ühe päringu kohta. Ligi 15. minuti juures hakkas keskmine latentsus stabiilselt alla nelja sekundi jääma ning kuni testi lõpuni ligines aeglaselt 3 sekundi märgini. Selline latentsus on minu arvates päris halb arvestades seda, et see platvorm peaks automaatselt suutma ressursse juurde luua, et selliste päringute arvuga toime tulla. Negatiivne on veel lisaks sellele see, et eriti kõrgeid väärtuseid oli üsna palju. Positiivse poole pealt on näha, et vähemalt äärmuslikke latentsusväärtusi muutus vähemaks aja möödudes, mis võib olla mingi märk sellest, et mingeid ressursse ikkagi eraldati funktsioonile juurde, kuid mitte piisavalt, et tagada arvestatav latentsus lihtsa registreerimispäringu jaoks. Teisalt, kui vaadata päringute õnnestumisi ja ebaõnnestumisi jooniselt 28**,** siis sealt on näha, et ainult 0.11% päringutest ebaõnnestus, mis on üpriski hea näitaja.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Register user	335147	5112	327	1029883	10332.65	0.11%	151.2/sec	83.19	58.12	563.3
TOTAL	335147	5112	327	1029883	10332.65	0.11%	151.2/sec	83.19	58.12	563.3

Joonis 28. Koormustesti päringute kokkuvõte registreerimise kasutuslool AWS Lambda platvormil.

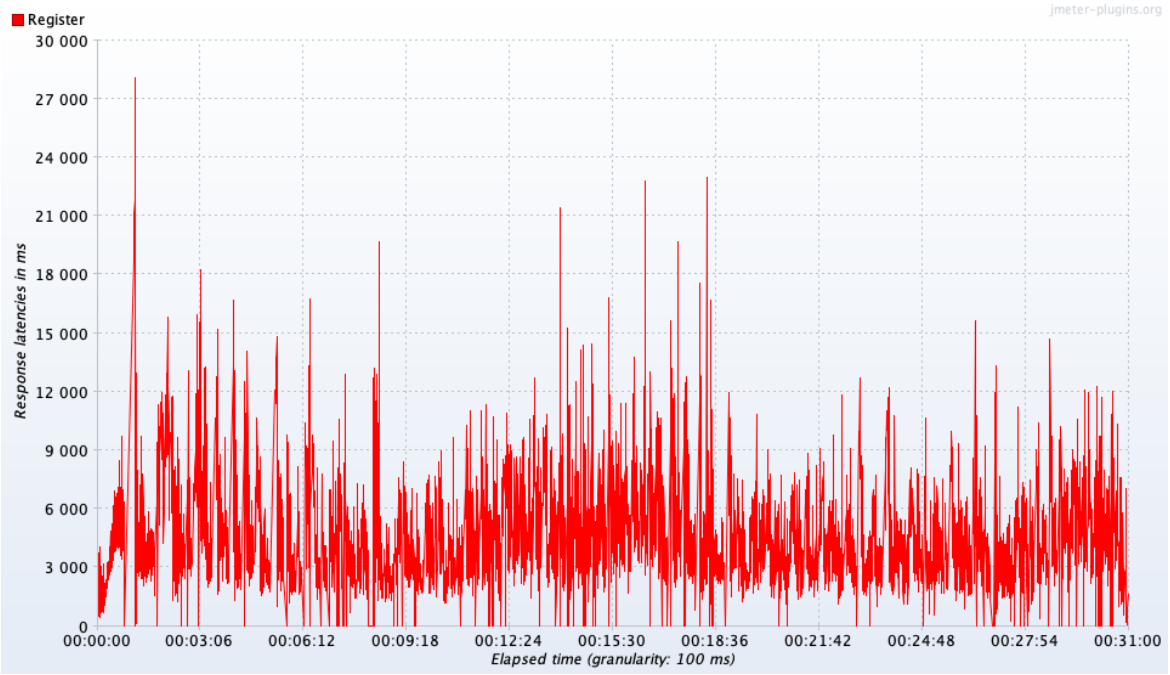


Joonis 29. Koormustesti latentsuste graafik registreerimise kasutuslool AWS Lambda platvormil.

IBM OpenWhisk platvormil oli juba raske toime tulla pikaajalise testi koormustega ning seda enam oli see ilmne koormustestil, kus päringute arv oli mitu korda suurem. Seda ilmestab joonis 30, kus on näha, et 82.26% päringutest sai vastuseks veateate, mille põhjuseks oli logide järgi andmebaasi kirjutamise ja lugemise piirangud. Üldiselt on näha joonisel 31 oleval latentsuste graafikust, et tegelikult püsis latentsus enamvähem stabiilne, kuid eelkõige oli probleeme funktsioonil, siis kui tulid esialgsed suured päringute arvud. Seoses sellega, et andmebaas andis nii mitmel juhul veateate võis ka latentsust mõjutada, et selle võrra pidi vähem infot töötlema ja tagastama, kuid keskmine latentsus tuli ikkagi 6286 ms mis on sekundi võrra rohkem kui AWS Lambda platvormil.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Register	359683	6286	76	775665	34231.58	82.26%	193.3/sec	157.44	78.36	833.9
TOTAL	359683	6286	76	775665	34231.58	82.26%	193.3/sec	157.44	78.36	833.9

Joonis 30. Koormustesti päringute kokkuvõte registreerimise kasutuslool IBM OpenWhisk platvormil.

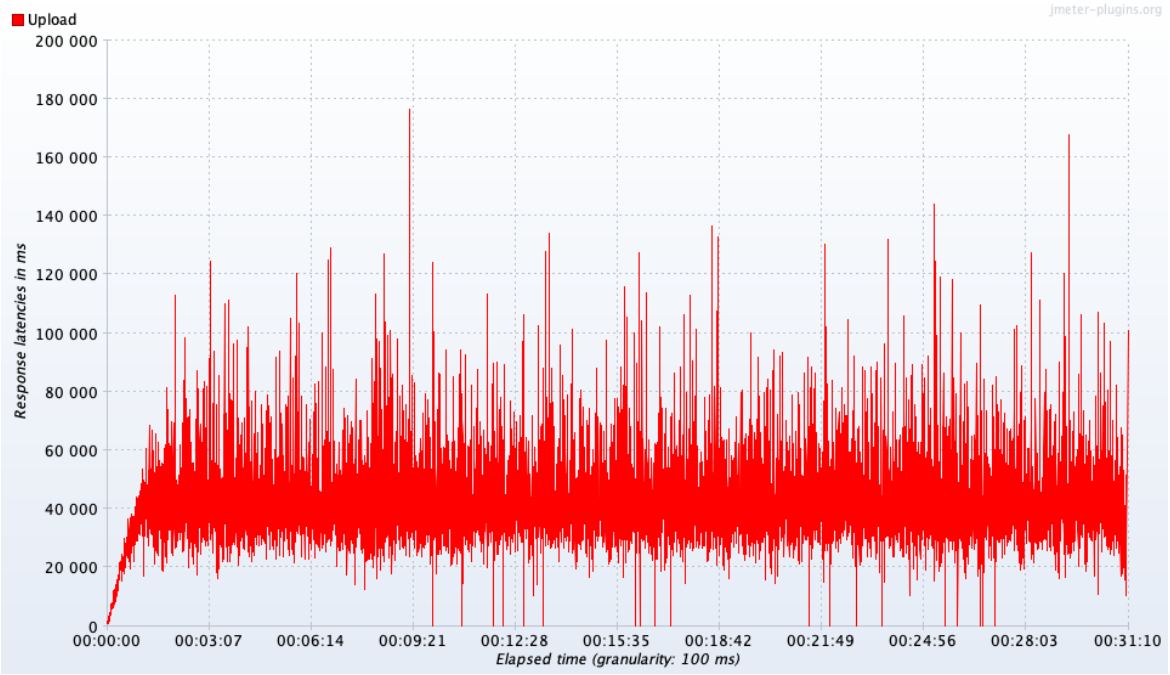


Joonis 31. Koormustesti latentsuste graafik registreerimise kasutuslool IBM OpenWhisk platvormil.

Teise kasutusloo puhul AWS Lambda platvormil on näha joonisel 33, et latentsused püsisid reeglina stabiilsena, kuid mõningate tugevate ekstreemumitega. Jooniselt 32 on näha, et keskmine latentsus oli 42359 ms, mis teeb ligikaudu 42.4 sekundit ning veateateid saanud päringute arv oli 0.33% koguarvust. Võrreldes esimese kasutuslooga, siis on siin latentsus mitmekordne ning oleks oodanud FaaS platvormide puhul lubatud skaleeruvust, kuid seda ei juhtunud.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Upload	21742	42359	688	247494	15135.65	0.33%	11.6/sec	5.54	2236.31	487.7
TOTAL	21742	42359	688	247494	15135.65	0.33%	11.6/sec	5.54	2236.31	487.7

Joonis 32. Koormustesti päringute kokkuvõte pildi üleslaadimise kasutuslool AWS Lambda platvormil.

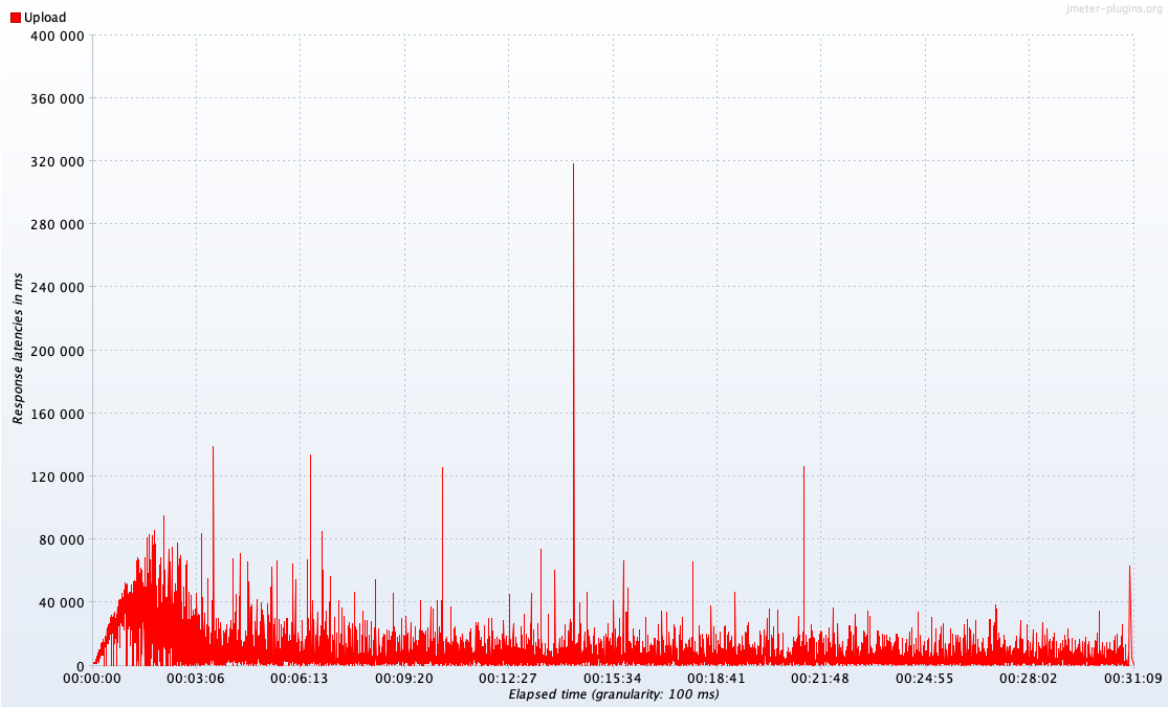


Joonis 33. Koormustesti latentsuste graafik pildi üleslaadimise kasutuslool AWS Lambda platvormil.

Sama testi tehes IBM OpenWhisk platvormil on joonisel 34 näha, et veateadetega päringuid oli vähem kui registreerimise kasutuslool ehk ainult 4.87% ning selle põhjuseks on see, et ei toimunud suhtlust andmebaasiga. Teisalt kui võrrelda seda AWS Lambda platvormiga, siis on näha, et see on ikkagi märkimisväärselt kõrgem. Samas oli selle testi käigus keskmine latentsus 27792 ms parem kui AWS Lambdal ning seda ligi 15000 ms võrra, mis on üpris palju. Samuti on joonisel 35 olevalt latentsuste graafikust näha, et IBM OpenWhisk kohandas ennast päringute arvule ning latentsus muutus stabiilselt madalamaks testi käigus.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Upload	12967	27792	569	819416	64704.88	4.87%	6.9/sec	2207.39	1298.41	325954.0
TOTAL	12967	27792	569	819416	64704.88	4.87%	6.9/sec	2207.39	1298.41	325954.0

Joonis 34. Koormustesti päringute kokkuvõte pildi üleslaadimise kasutuslool IBM OpenWhisk platvormil.



Joonis 35. Koormustesti latentsuste graafik pildi üleslaadimise kasutuslool IBM OpenWhisk platvormil.

4.4.Analüüs

Analüüsi alusteks on eelnevalt paika pandud kriteeriumid, milleks olid keskkonna ülesseadmine, latentsus külmkäivitamisel ja tavalise käivitamisel ajaperioodi jooksul, latentsus pikema aja jooksul, koormustest.

4.4.1. Keskkonna ülesseadmine

Kasutusmugavuse ja -lihtsuse osas võeti arvesse seda, et keskkondade ülesseadjal puudub eelnev kogemus pilveplatvormidega ning sellest tulenevalt on paremaks platvormiks AWS Lambda ning seda mitmel põhjusel. Kõige olulisemaks pean ma seda, et kui lihtne on probleemide tekkimisel leida abi või leida juhendeid erinevate funktsionaalsuste arendamistel. Selle põhjuseks on ilmselt see, et kuna Amazon on pilveteenuste turuliider, mis tähendab seda, et omakorda on kasutajabaas on niivõrd palju suurem ja sellest tulenevalt on väga suur tõenäosus, et keegi on mingi probleemiga juba silmitsi seisnud.

Teisalt räägib IBMi kasuks tema vähesemad funktsionaalsused, mistõttu on tema platvormist kergem aru saada ning ka probleeme võib vähem tekkida. Üheks näiteks on see, et IBM platvormil ei ole vaja mingit vahekihti selleks, et funktsioone välja kutsuda, kuid AWS Lambda puhul oli seda vaja API Gateway näol. Sellest tulenevalt tekkis funktsioonide loomise ja väljakutsumise käigus probleeme, mida IBM OpenWhisk teenuses ei tekkinud.

Näiteks oli üsna keeruline seadistada API Gateway teenust tagastama binaarset meediatüüpi ehk teise kasutusloo pilti, kuid see-eest suutis IBM seda teha automaatselt. Sama võis täheldada ka esimese kasutusloo puhul, kus AWS Lambda platvormil pidi sisendit veel teisendama. Teisalt tähendab see seda, et API Gateway teenust on võimalik rohkem enda kasutusloo järgi kohendada, kuid see tuleb selle arvelt, et on vaja fundamentaalselt aru saada, kuidas API Gateway ja AWS Lambda omavaheline suhtlus toimib.

Mis puudutab mõlema võrrelda platvormi dokumentatsioone, siis ma ütleks, et ka selles kategoorias on paremaks platvormiks Amazon, sest praktiliselt iga kasutusloo ja integratsiooni jaoks on põhjalik dokumentatsioon olemas. Samuti märkasin ka seda, et Amazoni enda töötajad on aktiivsed näiteks StackOverflow platvormil, kus nad abistavad kasutajaid küsimuste tekkimisel. IBM platvormi dokumentatsioon oli ka hea ning aitas lihtsamate küsimuste põhjal, kuid selle põhjalikkust ei anna võrrelda teise platvormiga.

Kahe võrreldava platvormi ühiseks negatiivseks küljeks on see, et koodi on raske siluda ning peab seda tegema ilma spetsiaalse tarkvarata. See tähendas seda, et koodi silumiseks oli kaks varianti – printida funktsiooni keskel mingi väärtus välja ning seda hiljem logidest vaadata või veateate puhul uurida logidest veateadet. Juhul kui veateadet ei olnud, siis tuli pöörduda esimese variandi poole. Sellest tulenevalt võib olla raskendatud selliste funktsioonide silumine, mis on omakorda seotud teiste funktsioonidega.

4.4.2. Testide tulemused

Külmkäivituse test näitas, et see külma funktsiooni esmakordne väljakutsumise probleem on püsiv, kuid väiksemate koormuste ning vähem arvutusressurssi nõudvate funktsioonide puhul ei ole see nii ilmseks probleemiks. Suuremaks probleemiks oli see arvutusressurssi nõudvate funktsioonide korral, kus latentsused olid väga ebastabiilsed mõlemal platvormil. Teisalt tuleb siin arvesse võtta ka seda, et arvutusressurssi nõudvatele platvormidele oli eraldatud ka rohkem mälu, mis võis tingida selle, et instantsi loomine võttis kauem aega ning samuti võis funktsioonide latentsust mõjutada ka testarvuti enda jõudlus, kuid see ei seletaks suuri latentsuse vahesid AWS Lambda ja IBM OpenWhisk vahel.

Pikaajalise testi tulemuste põhjal võib öelda seda, et ka väiksemate koormuste korral pika aja jooksul on IBMi platvormil suured raskused skaleeruvusega ning seda olenemata sellest, kas kasutatakse vahepeal suhtlust andmebaasi platvormiga või mitte.

Sama muster kordus ka koormustesti korral registreerimise kasutuslooga IBM platvormil, kuid samas oli näha, et teise kasutusloo puhul olid tulemused märkimisväärselt paremad

ning edestasid vastava testi tulemusi AWS Lambda platvormil. Sama test IBM platvormil näitas ka seda, et IBM platvormil olev funktsioon suutis kohaneda päringute arvuga, kuid AWS Lambda platvormi puhul oli näha, et latentsus püsis stabiilsena.

Üldiselt võib testide põhjal väita, et IBM OpenWhisk platvormil oli probleeme andmebaasi skaleeruvusega, sest veateadete arv registreerimise kasutusloo puhul igas testis oli väga suur võrreldes AWS Lambda olematute veateadete arvuga. Seega võib väita, et AWS platvormi kasutatud DynamoDB andmebaas suudab paremini skaleeruda kui IBM Cloud platvormil pakutav Cloudant andmebaas, sest mõlemal juhul oli valitud vastav andmebaasi plaan, mille ressursid olid varuga võrreldes testide koormustega. Sellest tulenevalt võiks FaaS platvormidest rääkides arvestada neid toetavate andmebaaside skaleerumise võimest, sest reaalsuses kasutavad väga paljud erinevad kasutuslood andmebaasiga suhtlust. See sama toodi ka välja seotud töödes, kus öeldi, et FaaS platvorme hoiab tagasi nende andmebaaside skaleeruvus.

Samuti oli testidest näha, et külmkäivitamine reeglina ei olnud probleemiks väikeste päringute mahu ja mitte arvutuslikult võimsate tegevuste korral, kuid suuremate päringu hulgaga ning arvutusmahukama funktsiooni korral oli näha. Samuti oleks oodanud rohkem näha automaatset skaleeruvust, sest see on üks nendest funktsionaalsustest, mille põhjal peaks olema FaaS platvormid etemad teistest lahendustest. Mingil määral oli seda testides näha, kuid latentsused jäid ikkagi suhteliselt kõrgeks.

5. Kokkuvõte

Käesolev bakalaureusetöö andis ülevaate viimasest tehnoloogilisest edasiarendusest pilveplatvormidel, milleks on pilveteenustes pakutavad funktsioonid. Selle töö eesmärgiks oli anda ülevaade populaarsematest pilveteenusepakkujate FaaS platvormidest ning võrrelda praktiliste testide abil AWS Lambda ja IBM Cloud Functions platvorme. Need platvormid said valitud seetõttu, et AWS on pilveteenuste turuliider ning IBM Cloud Functions põhineb vabavaralisel Apache OpenWhisk platvormil.

Praktilise võrdluse käigus selgus, et seotud töödes märgitud probleemid nagu külmkäivitus ja suhtlus andmebaasidega on probleemiks ning on limiteerivaks faktoriks FaaS platvormide kasutuselevõtul. Üldiselt oli näha, et külmkäivitus mõjutas tööd eriti arvutusmahukamatel funktsioonidel ning see mõjutas ka funktsiooni jõudlust ka edaspidi, kuid vähem ressurssi nõudvate ülesannete puhul see nii suureks probleemiks ei olnud. Koormustestide korral suutsid mõlemad platvormid testide lõpuks keskmist latentsust stabiilselt vähendada, kuid mitte arvestatavalt ning keskmine latentsus jäi siiski võrdlemisi kõrgeks.

Testide põhjal võib veel väita, et üldiselt esines AWS Lambda platvorm paremini ning eelkõige sellepärast, et igas läbi mängitud stsenaariumis oli veateate saanud päringute arv minimaalne. IBM OpenWhisk platvormil oli väga suuri probleeme andmebaasi skaleeruvusega, mis põhjustas ka selle, et väga paljud päringud registreerimise kasutuslool veateate.

Tehtud praktilisi võrdlusi saaks täiendada veel võrdlustega teiste platvormidega nagu Google Cloud Functions ja Microsoft Azure Functions platvormidega. See annaks aimduse, kas siin töös välja toodud probleemid on aktuaalsed ka teistel platvormidel.

6. Kasutatud kirjandus

[1] S. Bhardwaj, L. Jain ja S. Jain, „Cloud computing: A study of infrastructure as a service (IAAS).," *International Journal of engineering and information Technology,* kd. 2, nr 1, pp. 61-62, 2010.

[2] E. Jonas, A. Khandelwal, K. Krauth, J. Schleier-Smith, Q. Pu, N. Yadwadkar, I. Stoica, V. Sreekanti, V. Shankar, J. E. Gonzalez, D. A. Patterson, C.-C. Tsai, J. Carreira ja R. A. Popa, „Cloud Programming Simplified: A Berkeley View On Serverless Computing," UC Berkeley, 2019.

[3] M. Roberts, „Serverless Architectures," 22 5 2018. [Võrgumaterjal]. Available: https://martinfowler.com/articles/serverless.html. [Kasutatud 8. aprill 2019].

[4] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleicher-Smith, V. Sreekanti, A. Tumanov ja C. Wu, „Serverless Computing: One Step Forward, Two Steps Back," UC Berkeley, California, 2019.

[5] V. Ishakian, I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski ja P. Suter, „Serverless Computing: Current Trends and Open Problems," Faculty of Distributed, Parallel, and Cluster Computing, 2017.

[6] Amazon Web Services, „AWS Documentation," 2019. [Võrgumaterjal]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. [Kasutatud 24. märts 2019].

[7] Amazon Web Services, „AWS Step Functions," Amazon, 2019. [Võrgumaterjal]. Available: https://aws.amazon.com/step-functions/. [Kasutatud 9. aprill 2019].

[8] Amazon Web Services, „AWS Documentation," 2019. [Võrgumaterjal]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html. [Kasutatud 24. aprill 2019].

[9] Amazon Web Services, „AWS Lambda Pricing," 2019, [Võrgumaterjal]. Available: https://aws.amazon.com/lambda/pricing/. [Kasutatud 13. aprill 2019].

[10] Google Cloud, „Writing Cloud Functions," Google, [Võrgumaterjal]. Available: https://cloud.google.com/functions/docs/writing/. [Kasutatud 15. aprill 2019].

[11] Google, „Google Firebase," Google, 2019. [Võrgumaterjal]. Available: https://firebase.google.com/. [Kasutatud 15. aprill 2019].

[12] Google Cloud, „Google Cloud Functions Pricing," Google, [Võrgumaterjal]. Available: https://cloud.google.com/functions/pricing. [Kasutatud 15. aprill 2019].

[13] IBM Cloud, „IBM Cloud Functions Getting Started," IBM, 15. jaanuar 2019. [Võrgumaterjal]. Available: https://console.bluemix.net/docs/openwhisk/index.html#index. [Kasutatud 8. aprill 2019].

[14] IBM Cloud, „Cloud Functions Technology," IBM, 15. jaanuar 2019. [Võrgumaterjal]. Available: https://console.bluemix.net/docs/openwhisk/openwhisk_about.html#technology. [Kasutatud 8. aprill 2019].

[15] M. Mudrinic, „How To Install and Secure OpenFaaS Using Docker Swarm on Ubuntu 16.04," DigitalOcean, 18. september 2018. [Võrgumaterjal]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-openfaas-using-docker-swarm-on-ubuntu-16-04. [Kasutatud 16. aprill 2019].

[16] Apache OpenWhisk, „Apache OpenWhisk Open Source Serverless Cloud Platform,“ Apache, [Võrgumaterjal]. Available: https://openwhisk.apache.org. [Kasutatud 15. aprill 2019].

[17] Apache OpenWhisk, „Apache OpenWhisk Documentation,“ Apache, [Võrgumaterjal]. Available: https://openwhisk.apache.org/documentation.html. [Kasutatud 15. aprill 2019].

[18] Canalys, „Cloud Market Share Q4 2018 and Full Year 2018,“ 4 2 2019. [Võrgumaterjal]. Available: https://www.canalys.com/newsroom/cloud-market-share-q4-2018-and-full-year-2018. [Kasutatud 29. aprill 2019].

[19] Y. Cui, „How Long Does Aws Lambda Keep Your Idle Functions Around Before a Cold Start?,“ 4 7 2017. [Võrgumaterjal]. Available: https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810. [Kasutatud 2. mai 2019].

[20] J. Pokorny, „NoSQL Databases: a step to database scalability in Web environment,“ *International Journal of Web Systems,* kd. 9, nr 1, pp. 69-82, 2013.

[21] Amazon Web Services, „Amazon DynamoDB - Overview,“ 2019. [Võrgumaterjal]. Available: https://aws.amazon.com/dynamodb/. [Kasutatud 29. aprill 2019].

[22] Amazon Web Services, „What Is Amazon API Gateway?,“ [Võrgumaterjal]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html. [Kasutatud 29. aprill 2019].

I. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Mark-Eerik Kodar**,
(*autori nimi*)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose
FaaS pilveplatvormide võrdlus,
(*lõputöö pealkiri*)

mille juhendaja on Pelle Jakovits,

(*juhendaja nimi*)

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

1. 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 autoriõiguse kehtivuse lõppemiseni.

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

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

Tartus, **10.05.19**