TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kristjan Lilleoja 162860IABM

MONOLIIDI MIGREERIMINE MIKROTEENUSTEL PÕHINEVALE SERVERIVABALE ARHITEKTUURILE

Magistritöö

Juhendaja: Tarmo Robal
PhD

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Kristjan Lilleoja

07.05.2019

Annotatsioon

Käesoleva magistritöö põhieesmärgiks on uurida ja analüüsida mikroteenustel põhineva serverivaba arhitektuuri kasutamise eeliseid ja puudusi võrreldes monoliitse arhitektuuriga. Põhieesmärki toetava eesmärgina kirjeldatakse küsitlustarkvara Surveer varasemat arhitektuuri ja dokumenteeritakse serverivaba arhitektuuri kasutuselevõttu, migreerides Surveeri kõrgema äririskiga monoliitse REST API otspunktid serverivabadeks mikroteenusteks, kasutades selleks Serverless raamistikku koos AWS Lambda platvormiga.

Serverivaba arhitektuuri (keskendudes FaaS'ile) eelisteks leiti suurem produktiivsus, automaatne skaleerimine, kulude kokkuhoid ja „rohelisem" andmetöötlus. Peamisteks puudusteks osutusid tootjalukustus, turvalisusküsimused, funktsioonide oleku puudumine, külmkäivitus ning piiratud käitusaeg ja mälumaht. Jõudlusanalüüsi tulemustes näitasid mõlemad arhitektuurid tippkoormustes võrdväärset reaktsiooniaega, kuid erinevus seisnes tippkoormusel AWS Lambda stabiilsemas skaleeruvuses. Kuluanalüüsi tulemuste põhjal selgus, et kahe arhitektuuri võrdluses oli traditsiooniline arhitektuur Amazon EC2 koos Application Load Balancer'iga (ALB) kulutõhusam alates koormusest 7,7 päringut sekundis. Kuid kõrvutades eelnevatega kolmandat võimalikku serverivaba arhitektuuri, milles AWS Lambda platvormi kasutatakse Amazon API Gateway asemel ALB'iga, ilmnes, et koormusel 0 kuni 2 päringut sekundis oli kulutõhusam Lambda koos API Gateway'ga, aga alates kahest päringust sekundis on selleks Lambda + ALB.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 52 leheküljel, 6 peatükki, 13 joonist, 8 tabelit.

Abstract

Migration of Monolith to Microservice-Based Serverless Architecture

The main aim of this thesis is to study and analyse the benefits and drawbacks of microservice-based serverless architecture compared to traditional monolithic architecture. The subsidiary aim is to describe and document both the earlier monolithic architecture and the new serverless architecture of online survey software Surveer, migrating REST API endpoints with higher business risk into serverless microservices using the Serverless Framework with AWS Lambda platform.

The thesis includes an overview of related research in the subject area, which showed great interest growth in the serverless architecture research field and referred to the positive results in various studies. The found benefits of serverless architecture (focusing on FaaS) are higher productivity, auto-scaling, reduced cost and "greener" computing. The main drawbacks turned out to be vendor lock-in, security concerns, no state for FaaS-functions, cold starts, execution time and memory limits.

To achieve the aim, research questions were defined in order to evaluate the API endpoints performance and to compare costs between earlier monolithic and new serverless architecture. Answers are found using a series of performance testing that compare Surveer's earlier and new architecture based on a common test plan.

Performance analysis results showed similar response times at peak loads for both architectures, but the difference was AWS Lambda's more stable scalability. The results of cost analysis showed that comparing the two architectures, traditional Amazon EC2 with Application Load Balancer (ALB) architecture is more cost-effective starting from 7.7 requests per second. This changes when a third serverless architecture is added to the comparison, in which the AWS Lambda platform is used with ALB. In that case, at 0 to 2 queries per second the most cost-effective architecture is Lambda with Amazon API Gateway but from 2 queries per second, it is AWS Lambda with ALB.

The thesis is in Estonian and contains 52 pages of text, 6 chapters, 13 figures, 8 tables.

Lühendite ja mõistete sõnastik

Ajax	Asünkroonne JavaScript ja XML (ingl *Asynchronous JavaScript And XML*)
AngularJS	JavaScript raamistik eesrakendustele
ALB	Application Load Balancer – AWS'i uut tüüpi Elastic Load Balancing koormuse tasakaalustaja
API	Rakendusliides (ingl *Application Programming Interface*)
API Gateway	Amazon API Gateway – AWS'i teenus REST ja WebSocket API-de loomiseks, avaldamiseks, hooldamiseks, seireks ja kindlustamiseks
AWS	Amazon Web Services – populaarne pilveteenuse pakkuja
CloudFront	Amazon CloudFront – AWS'i sisuedastusvõrk (ingl *Content Delivery Network*, CDN)
CSS	Kaskaadlaadistik, HTML-dokumendi ilmet määravate laadilehtede hierarhiline süsteem (ingl *Cascading Style Sheets*)
DBaaS	Andmebaas teenusena (ingl *Database as a Service*)
EC2	Elastic Compute Cloud ehk Amazon EC2 – AWS'i toode, mis pakub skaleeritavat andmetöötlust pilves
Express	Node.js raamistik
FaaS	Serverivaba tehnoloogia, funktsioon teenusena (ingl *Function as a Service*)
Git	Vaba versioonihaldustarkvara
GitHub	Git repositooriumide veebipõhine majutusteenus
HTML	Hüperteksti märgistuskeel (ingl *HyperText Markup Language*)
HTTP	Hüperteksti edastusprotokoll (ingl *HyperText Transfer Protocol*)
HTTPS	Turvaline hüperteksti edastusprotokoll (ingl *HyperText Transfer Protocol Secure*)
IaaS	Infrastruktuur teenusena (ingl *Infrastructure as a Service*)
JMeter	Apache JMeter – avatud lähtekoodiga tarkvara veebirakenduste koormustestide tegemiseks ja jõudluse mõõtmiseks
JSON	Lihtsustatud andmevahetusvorming, mis põhineb JavaScript'i programmeerimiskeele alamhulgal (ingl *JavaScript Object Notation*)
Lambda	AWS Lambda – AWS'i FaaS-teenusplatvorm
Lambda@Edge	AWS Lambda funktsioonid CloudFront sisuedastusvõrgus

MongoDB	Platvormist sõltumatu vabavaraline dokumendi-põhine andmebaasi haldamise süsteem
MEAN	Akronüüm tehnoloogiatele MongoDB, Express, AngularJS ja Node.js
Nginx	Veebiserverite tarkvara
Npm	Node.js'i vaikimisi paketihaldur
Node.js	JavaScript'il põhinev serveri platvorm
PaaS	Platvorm teenusena (ingl *Platform as a Service*)
PHP	Veebiprogrammeerimiskeel (ingl *Hypertext Preprocessor*)
Python	Üldotstarbeline interpreteeritav programmeerimiskeel
REST	Programmiarhitektuuri stiil ja suhtlusviis ees- ja tagarakenduse vahel (ingl *Representational State Transfer*)
Route 53	Amazon Route 53 – AWS'i domeeninimede süsteemi (ingl *Domain Name System*, DNS) teenus
SaaS	Tarkvara teenusena (ingl *Software as a Service*)
Serverless	Vabavaraline raamistik serverivabade rakenduste arendamiseks ja juurutamiseks
S3	Amazon Simple Storage Service (Amazon S3) – AWS'i objektipõhine andmesalvestusteenus
Stack Overflow	Küsimuste ja vastuste portaal programmeerijatele
URL	Internetiaadress ehk universaalne ressursilokaator (ingl *Uniform Resource Locator*)

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Käesolevat ajajärku ajaloos kirjeldatakse kui infoajastut, mida iseloomustab kiire üleminek traditsiooniliselt tootmiselt, mille tööstusrevolutsioon kaasa tõi, infotehnoloogial põhinevale majandusele. Maavarade asemel on saanud majanduse peamisteks ressurssideks informatsioon ja infotehnoloogia. Tänapäeval toimub ettevõtete väärtusloome suuresti neid ressursse kasutades. Üks olulisemaid infotehnoloogilisi vahendeid, mille läbi informatsiooni liikumine toimub, on veebiserver.

Maailma esimene veebiserver oli NeXT arvuti NeXTSTEP operatsioonisüsteemi ja vastloodud CERN httpd serveritarkvaraga, mille autorid olid Luotonen, Nielsen ja Berners-Lee [1]. Alates veebi loomisest 1990ndate aastate alguses, on see läbi teinud plahvatusliku arengu. Netcraft'i poolt korraldatud uuringu kohaselt oli 2018. aasta augustis Internetis 1 661 467 123 veebisaiti, 221 524 704 unikaalset domeeni ja 7 758 309 veebile avatud serverit [2]. Seda kasvu peegeldab ka veebiserverite ja -arhitektuuride areng.

Kui veeb sai alguse üksikutest võrku ühendatud tööjaamadest, siis üsna pea asendusid need sihtotstarbeliste serveritega. Aja möödudes majutati need eraldi serveriruumidesse, moodustades esimesed andmekeskused. Kuna see nõudis ettevõtetelt suuri investeeringuid ja operatsioonikulusid, siis hakati otsima võimalusi andmekeskuste ressursi targemaks kasutamiseks, sh võimalust neid mitme osapoole vahel jagada. Virtualiseerimise tehnoloogia areng tekitas selle võimekuse. Amazon.com lõi 2006. aasta augustis tütarettevõtte Amazon Web Services (AWS), tuues turule teenuse Amazon Elastic Compute Cloud (EC2) [3], mis populariseeris uue paradigma – pilvandmetöötluse (ingl *cloud computing*).

Pilvandmetöötluse laialdase kasutuselevõtuga on pilveteenuse pakkujad muutumas riikide majanduste jaoks üha olulisemaks. Nad majutavad märkimisväärsel hulgal nende ettevõtete missioonikriitilist äritarkvara, mis enam ei kasuta (või pole kunagi kasutanud) omi andmekeskusi [4]. Weinman on esitanud matemaatilise tõestuse, et enamiku reaalsete töökoormuste puhul vähendab pilvandmetöötluse kasutuselevõtt kulusid võrreldes ainult

majasiseste serverite kasutamisega [5]. Sellest lähtuvalt on pilvandmetöötlus avaldanud suurt mõju ka veebirakenduste arhitektuuridele, mis üha enam sõltuvad pilveteenuse pakkujatest. Vahepealne virtualiseerimise areng, täpsemalt konteinerite tehnoloogia võidukäik, on võimalikuks teinud järjekordse murrangu – serverivaba andmetöötluse (ingl *serverless computing*).

Serverivaba andmetöötlus, tuntud ka kui funktsioon teenusena (ingl *Function as a Service*, FaaS), on pilverakenduste juurutamise uus veenev paradigma, milles ühe monoliitse teenuse asemel jagatakse ja juurutatakse rakenduse loogika eraldiseisvate funktsioonidena (mikroteenustena). Pilveteenuse tarbijal "puuduvad" serverid, mida administreerida (see on teenusepakkuja vastutusel), platvormi tuleb laadida üksnes kood. Pilveteenuse pakkuja vastutab konteinerite-põhise paralleelse automaatse skaleerimise eest, seega käideldavus on tagatud nii ühe kui ka tuhande samaaegse käivituse puhul. Ning viimaks, maksma peab ainult kasutatud ressursside eest. AWS hakkas FaaS-teenusplatvormi võimalust esmakordselt pakkuma 2014. aasta lõpus AWS Lambda[1] nime all. Käesolevaks ajaks on FaaS-teenusplatvormidega välja tulnud lisaks AWS'ile ka teised suured pilveteenuste pakkujad, nagu Google Cloud[2], Microsoft Azure[3] ja IBM Cloud[4].

Autori bakalaureusetöö[5] raames loodud tarkvara Surveer[6] on veebipõhine rakendus küsitluste ja uuringute loomiseks. Rakendus realiseeriti algselt MEAN-platvormil – MongoDB andmebaasi, monoliitse Node.js REST (ingl *Representational State Transfer* – programmiarhitektuuri stiil) API (ingl *Application Programming Interface* ehk rakendusliides) ja AngularJS eesrakendustega. Vahepeal on populaarsust kogunud serverivaba arhitektuur, mille kasutuselevõtt võib lahendada rakendusega seotud probleeme.

Peamine esilekerkinud probleem seisneb selles, et kuna küsitlustele vastajate arvu on keeruline ette ennustada ning see kasvab ja kahaneb hüppeliselt, on serveriressursside skaleerimine osutunud tülikaks ning kõrgkäideldavuse saavutamiseks peab tasuma ka

[1] https://aws.amazon.com/lambda/
[2] https://cloud.google.com/functions/
[3] https://azure.microsoft.com/en-us/services/functions/
[4] https://console.bluemix.net/openwhisk/
[5] https://bitbucket.org/kristjanlilleoja/thesis/raw/master/archive/Kristjan_Lilleoja_120407_BAK.pdf
[6] https://surveer.com/

nende ressursside eest, mida enamjaolt ei kasutata. Samuti võib skaleerimise probleemidega tekkida rakenduse maasolekuaja oht, mille korral võib realiseeruda oluline äririsk – kliendid kaotavad potentsiaalsed andmed, kuna küsitlustele ei saa vastata. Seega on oluline uurida, kas serverivaba arhitektuuri kasutuselevõtt võimaldaks neid probleeme lahendada.

Käesoleva magistritöö põhieesmärk on uurida ja analüüsida mikroteenustel põhineva serverivaba arhitektuuri kasutamise eeliseid ja puudusi võrreldes monoliitse arhitektuuriga. Põhieesmärki toetavaks eesmärgiks on kirjeldada rakenduse Surveer varasemat arhitektuuri ja dokumenteerida serverivaba arhitektuuri kasutuselevõttu, migreerides Surveeri kõrgema äririskiga monoliitse REST API otspunktid serverivabadeks mikroteenusteks, kasutades selleks Serverless raamistikku koos AWS Lambda platvormiga. Töö tulemustest võiks olla abi ka teistele infosüsteemide arhitektidele ja analüütikutele otsuse langetamiseks serverivaba arhitektuuri kasutamise osas.

Töö eesmärgi saavutamiseks on püstitatud järgnevad uurimisküsimused:

1. Kuidas erineb serverivabale arhitektuurile migreeritud API-otspunktide jõudlus võrreldes monoliitse arhitektuuriga?

2. Millised on serverivaba arhitektuurile migreeritud API pilvekulude erinevused võrreldes monoliitse arhitektuuriga?

Uurimisküsimustele aitavad vastuseid leida eksperimentide seeriad, millega võrreldakse Surveeri varasemat ja uut arhitektuuri ühise testide komplekti tulemuste alusel.

Käesolev magistritöö on jaotatud kuuele peatükile. Peatükk 2 on ülevaade seotud teadustöödest. Peatükk 3 tutvustab serverivaba arhitektuuri, sh selle eeliseid ja puudusi ning erinevaid FaaS-teenusplatvorme. Peatükis 4 on kirjeldatud käesoleva juhtumiuuringu aluseks oleva küsitlustarkvara Surveer varasemat arhitektuuri, probleeme ja uue serverivaba arhitektuuri kasutuselevõttu. Peatükk 5 sisaldab uurimistulemuste analüüsi, tuues esile jõudlus- ja kuluanalüüsi. Peatükis 6 on esitatud töö kokkuvõte.

2 Ülevaade seotud teadustöödest

Pilvandmetöötlust on küll palju uuritud, kuid arvestades serverivaba tehnoloogia uudsust, pole antud valdkonna kohta akadeemilisi kirjutisi veel kuigi palju avaldatud, samas on märgata selles osas suurt kasvu. Näiteks otsides ACM digitaalsest raamatukogust töid, mille pealkirjas on termin „serverless“[1], selgub, et kui aastal 2014 ei avaldatud ühtegi kirjutist, siis aastal 2016 oli neid viis, aastal 2017 hüppeliselt 22 ning aastal 2018 avaldati 25 kirjutist. Kuigi akadeemiliste tööde koguarv kõnealuses teemavaldkonnas pole veel eriti suur, on siiski avaldatud mõned olulised uurimused, mis seostuvad lähedalt käesoleva magistritöö uurimisküsimuste ja eesmärkidega.

Adzic (Neuri Consulting) ja Chatley (Imperial College London) [6] uurisid serverivaba tehnoloogia majanduslikku ja arhitektuurilist mõju kahe juhtumiuuringu näitel – MindMup ja Yubl. MindMup[2] on kaubanduslik veebirakendus mõttekaartide loomiseks. Juhtumiuuringus toodi välja kasutuskoormuse võrdlus 2016. aasta veebruari (enne AWS Lambda migratsiooni) ja 2017. aasta veebruari (pärast migratsiooni lõpuleviimist) vahel. Sel perioodil kasvas rakenduse aktiivsete kasutajate arv veidi üle 50%, kuid kulud vähenesid veidi alla 50%. Tulemuseks saadi kokkuhoid 66%, hoolimata sellest, et tol aastal lisandus mitmeid uusi teenuseid. Yubl oli praeguseks tegevuse lõpetanud sotsiaalmeedia äpp. Juhtumiuuringus soovis ettevõte serverivabale arhitektuurile migreerimisega kiirendada rakenduse uute funktsioonide juurutamist. Enne migratsiooni 2016. aasta aprillis juurutasid nad tarkvara uusi versioone toodangukeskkonda keskmiselt 4-6 korda kuus. Uuendatud meeskonnaga ja migratsiooniga AWS Lambda teenusplatvormile tõusis toodangukeskkonda juurutamine maikuus 60 korrani ja novembriks oli see juba enam kui 80 korda kuus. Seejuures jäi tarkvara meeskonna suurus samaks (6 tarkvarainseneri). Artikli kokkuvõttes toodi välja, et serverivabad platvormid on tänapäeval kasulikud tähtsate ülesannete jaoks, kus kõige olulisem on kõrge läbilaskevõime, mitte madal latentsusaeg ja kus üksikuid päringuid saab täita suhteliselt lühikeses ajavahemikus. Autorite hinnangul on kulukokkuhoid taoliste serverivabade

[1] https://tinyurl.com/y9l5exse
[2] https://www.mindmup.com

funktsioonide puhul märkimisväärne ja lisaks toob serverivaba arhitektuur endaga kaasa suurema produktiivsuse ning kiirendab tarkvarauuenduste toodangukeskkonda juurutamist.

Crane ja Lin [7] uurisid empiiriliselt nende juhtumiuuringu aluseks oleva serverivaba arhitektuuriga otsingurakenduse jõudlust ja kulu. Autorite analüüs näitas, et kuigi esialgse teostatud lahenduse jõudlus jättis rakenduse eesmärgi (interaktiivne otsing) jaoks soovida, on serverivaba hinnastamismudel majanduslikult veenev ja tulevased infrastruktuuri parendused koos rakenduse optimeerimisega muudavad serverivaba arhitektuuri veelgi atraktiivsemaks. Artiklis esitatud käesoleva tööga lähedalt seostuvad eksperimendid olid seotud serverivaba arhitektuuri kulu analüüsimisega, milles oli võrdluse all rakendus Amazon EC2 (r3.4xlarge) pilveserveris ja AWS Lambda's. Nende kuluanalüüsi tulemuste põhjal oli Lambda tasuvuspunkt umbes 7,7 päringut sekundis, st Lambda oli kulutõhusam sellest väiksemal ja EC2 suuremal koormusel. Kuid nad rõhutasid fakti, et Lambda saavutab (potentsiaalselt piiramatu) skaleeruvuse ilma manuaalse sekkumiseta. Sellele tuginedes väljendasid artikli autorid arvamust, et kõikides rakendustes, välja arvatud kõige nõudlikumates (nt kaubanduslikud otsingumootorid), on serverivaba arhitektuur kulude vähendamise perspektiivist tulenevalt väga veenev.

Serverivaba arhitektuuriga seonduvaid kulusid analüüsisid ka Kolumbia teadlased [8], jõudes järeldusele, et mikroteenuste kasutamine võib aidata vähendada infrastruktuuri kulusid 13,42%. Kulude erinevust võrreldi monoliitse ja mikroteenustel põhineva arhitektuuri (nii serveriga kui ka serverivaba vormis AWS Lambda näol) vahel. Lisaks leidsid nad, et selliste uudsete FaaS-teenusplatvormide nagu AWS Lambda kasutamine, mis on kavandatud spetsiaalselt mikroteenuste kasutuselevõtuks granuleeritud tasemel (iga HTTP päringu/funktsiooni kohta), võimaldab ettevõtetel vähendada oma infrastruktuuri kulusid kuni 77,08%. Artikli kokkuvõttes toodi veel välja, et mikroteenused võimaldavad suurte rakenduste väljatöötamist väikeste rakenduste komplektides, mida saab iseseisvalt rakendada ja hallata, juhtides seega suuri koodibaase praktilisema metoodika abil, kus väikesed meeskonnad teevad täiendavaid parandusi üksteisest sõltumatutes koodibaasides.

Vastukaaluks eelmistele publikatsioonidele kirjutas The Walt Disney Company insenerilahenduste arhitekt Eivy teadusajakirjas IEEE Cloud Economics [9] mõnevõrra

vähem optimistliku artikli. Selles oli näide üksikust Node.js otspunktist, mida ta arendas ja mis tundus esmapilgul hea kandidaat AWS Lambda jaoks, kuni ta viis läbi kuluanalüüsi. Probleem oli prognoosis, mille järgi tema API't kutsutakse esialgu umbes 150 korda sekundis, eeldatava kasvu sihtmärgiga keskmiselt 30 tuhat päringut sekundis järgmisel aastal. Ta leidis, et kõnealuse rakenduse kulu oleks FaaS-teenusplatvormil võrreldes reserveeritud pilveserveriga kolm korda suurem. Rakenduse skaleeruvuse suurenemisel kasvas ka tohutult kulude erinevus võrreldes traditsioonilise arhitektuuriga. Need arvutused esitati kuu lõikes, mis tähendab, et aasta jooksul oleks kulude erinevus ligi pool miljonit dollarit – ainult ühe API otspunkti eest. Artikli autor kutsus üles hinnastamise vaatamisel olema ettevaatlik. Tema sõnul on petta saamise vältimiseks kriitiline arvestada nii funktsioonide planeeritud kui ka potentsiaalset skaleeruvust – et paremini mõista võimalikku kulu. Eivy hinnangul on kulude modelleerimine üsna keeruline, kuna peab arvestama kõike eelpool mainitut. Mitte iga rakendus ei saa samu tulemusi. Kui funktsioon käivitub kiiremini, tuleb maksta serveritulemuste eest vähem, kuna arvutamisaja maksumus on väiksem, aga kui see saab rohkem päringuid sekundis, siis kulu suureneb kiiresti. Võttes arvesse potentsiaalset aastase kulu erinevust, on artikli autori hinnangul arukas investeerida mõni tund või isegi mõni kuu, et modelleerida ja testida funktsioonide käivitamise aega mõistmaks paremini võimaliku arhitektuuri rahalisi tagajärgi. Kirjutis tõi lõpetuseks välja ühe praktilise nõuande. Nimelt on oluline kaaluda serverivaba arhitektuuriga API ligipääsu piiramist, et kaitsta ennast võimalike teenusetõkestamise (ingl *Denial-of-Service*, DoS) rünnete eest. Kui serverivabad funktsioonid on seotud avalikult ligipääsetava API'ga, ei tohiks unustada lisada päringute määra piiramist (ingl *rate limiting*). AWS Lambda't kasutades saab seda teha API Gateway teenuse abil. Vastasel juhul võib lihtne rünnak muutuda automaatse skaleerimise tõttu õudusunenäoks.

Serverivaba arhitektuuri kasutamist teaduslike uuringute töövoos uurisid Poola AGH Teaduse- ja tehnoloogiaülikooli teadurid [10]. Artiklis toodi välja seisukoht, et sellised infrastruktuurid, mis põhinevad serverivabade pilvefunktsioonide kontseptsioonil nagu AWS Lambda või Google Cloud Functions, pakuvad huvitavat alternatiivi mitte ainult tüüpilistele rakendustele, vaid ka teaduslikele töövoogudele. Samas leiti, et mitte kõik töökoormused ei sobi FaaS-funktsioonide maksimaalse eluaja tõttu, nt AWS Lambda puhul on see 15 minutit (oktoobris 2018 suurendati see varasemalt 5 minutilt [11]) – seega tuleb arvesse võtta ülesannete täpsust. Nende poolt koostatud kuluanalüüs

näitas, et serverivabade pilvefunktsioonide (FaaS) kasutamine võib olla kulutõhusam, kui traditsiooniliste IaaS (ingl *Infrastructure as a Service*) pilveteenuste kasutamine sõltuvalt rakenduse omadustest, näiteks ülesannete arvust ja täitmise tähtaegadest. Kuna see on suhteliselt uus teema, nähakse tulevaste tööde jaoks palju võimalusi.

Viimaks annab Kratzke uuriv artikkel [4] erinevate pilverakenduste (sh serverivaba) arhitektuuride osas pika ja põhjaliku ülevaate. Täpsemalt identifitseeris ja uuris ta kahte peamist trendi pilverakenduste arhitektuurides. Esiteks leidis Kratzke, et pilvandmetöötluse ja sellega seotud rakenduste arhitektuuride arengut võib pidada stabiilseks protsessiks ressursside kasutamise optimeerimisel. Teadlase hinnangul võib üks järgmine edasiarendus tulevikus rajaneda 1990. aastate lõpust pärit tehnoloogial, milleks on *unikernel*. Sarnaselt konteineritele[1] on need iseseisvad, kuid väldivad konteinerite ülekulu (ingl *overhead*), konteineri käitusmootorit ja isegi hosti operatsioonisüsteemi. Samas on artikli autori üllatuseks selles valdkonnas siiani tehtud vähe uurimistööd. Teiseks, iga uus ressursikasutuse parandamise viis on leidnud kasutust arhitektuuri evolutsioonis – kuidas pilverakendusi arendatakse ja juurutatakse. Täheldati nihet monoliitsest arhitektuurist, läbi iseseisvalt juurutatud mikroteenuste (konteinerite), serverivaba arhitektuuri (FaaS) suunas. Tema hinnangul on serverivabad arhitektuurid rohkem detsentraliseeritud ja jaotunud ning kasutavad tahtlikult enam iseseisvalt pakutavaid teenuseid. Üha rohkem on pilveteenuste orkestratsiooni loogika nihkumas lõppseadmetesse, mis on väljaspool teenuse haldamise süsteemi otsesest skoobist. Seega võivad domineerima asuda uued arhitektuurilised stiilid, nagu mikroteenused ja serverivabad arhitektuurid. Peale selle on märgata huvi taastumist detsentraliseeritud lähenemistele, mis on tuntud P2P-tehnoloogiaga (ingl *peer-to-peer*) seotud uurimistööst. See tundus Kratzke'le hämmastav, sest pilvandmetöötluse (tsentraliseeritud kontseptsioon) kasvu tõttu vähenes teadusuuringute huvi P2P põhiste lähenemisviiside (detsentraliseerimise kontseptsioon) vastu. Kuid samas näib see olevat muutumas ja võib viidata sellele, kuhu pilvandmetöötlus võiks tulevikus liikuda. Viimaks tõstatati Baldini jt [13] poolt esitatud küsimus: kas serverivaba ulatub kaugemale traditsioonilistest pilveplatvormidest? Arvestades hiljutisi suundumusi, tundub see Kratzke'le tõenäoline.

[1] Konteiner on standardne tarkvara ühik, mis pakendab koodi ja kõik selle sõltuvused nii, et rakendus töötab kiiresti ja usaldusväärselt erinevates arvutikeskkondades. Konteinerid jagavad operatsioonisüsteemi kernelit, mistõttu ei nõua nad eraldi operatsioonisüsteemi iga rakenduse kohta, suurendades seeläbi serveri efektiivsust ja vähendades serveri- ja litsentsimiskulusid [12].

Võib tõdeda, et viimastel aastatel on tehtud erinevaid uurimusi, mis seonduvad hästi käesoleva magistritöö uurimisküsimuste ja eesmärkidega. Uurimused näitavad suurt huvi kasvu serverivaba valdkonna vastu teadustöödes ning viitavad erinevates uurimistöödes saadud positiivsetele tulemustele serverivaba arhitektuuri osas. Arvestades serverivaba tehnoloogia uudsust, ei ole akadeemilisi uurimusi veel palju avaldatud, kuid samal ajal on valdkond ise koos erinevate teenusplatvormidega kiirelt arenenud. Seega on huvitav uurida, kas antud töö tulemused ühtivad varasemalt leituga.

3 Serverivaba arhitektuur

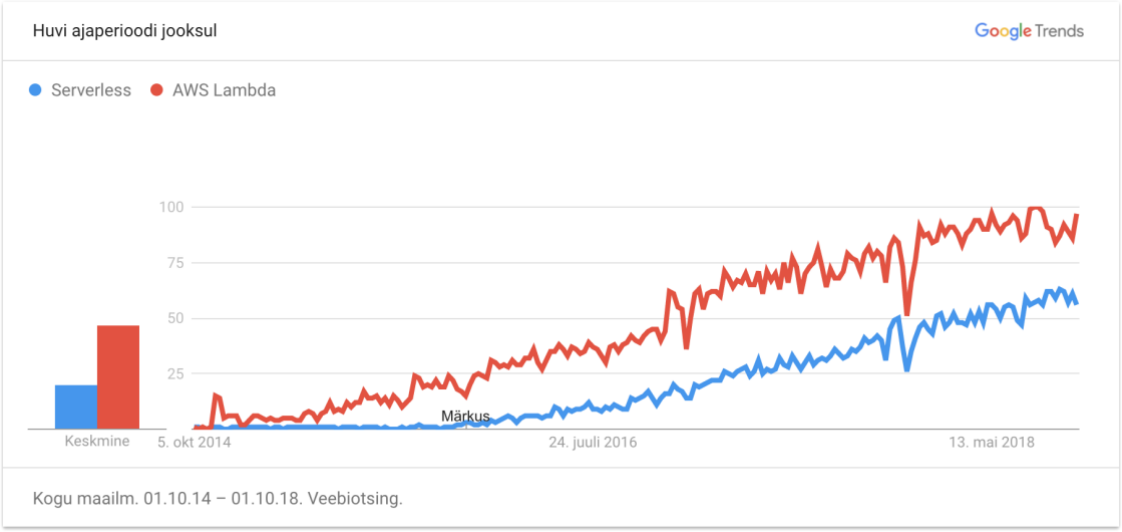
Termin „serverivaba“ (ingl *serverless*) on oma olemuselt veidi provokatiivne ja segadusttekitav, kuna sellise arhitektuuriga rakendustes on loomulikult kuskil olemas server [14]. Kuid erinevus võrreldes tavapärase lähenemisega on selles, et organisatsioon, mis arendab ja juurutab serverivaba arhitektuuriga rakendust, ei hoolitse sellega seotud riistvara ega süsteemiadministreerimise eest. See vastutus antakse üle kolmandale osapoolele.

Roberts on avaldanud põhjaliku ülevaate serverivabadest arhitektuuridest [14], tema hinnangul toimus „serverivaba“ termini esimene kasutus aastal 2012 avaldatud Fromm'i artiklis "*Why The Future Of Software And Apps Is Serverless*" [15], kuid laialdasem kasutuselevõtt algas pärast AWS Lambda käivitamist 2014. aastal ning sai veelgi populaarsemaks pärast Amazon API Gateway avalikustamist 2015. aasta juulis. Sama aasta oktoobris peeti Amazon'i „Re: Invent“ konverentsil ettekanne pealkirjaga "*The Serverless Company Using AWS Lambda*"[1] ning aasta lõpus muutis avatud lähtekoodiga projekt „*Javascript Amazon Web Services (JAWS)"* ennast Serverless raamistikuks [16], jätkates sama trendi. 2016. aasta keskpaigaks sai „serverivaba“ selle valdkonna domineerivaks nimetuseks, luues tee Serverlessconf[2] konverentside sarjale. Termin „serverivaba“ oli tulnud, et jääda.

Kuigi serverivaba andmetöötluse vanuseks võib lugeda kõigest viis aastat, on see kiirelt populaarsust kogunud ning saanud juba märgatavat tähelepanu. Ühest küljest näitab seda tihe konkureerivate teenuste juurdekasv pärast AWS Lambda debüüti 2014. aastal ja teisest küljest suur ülemaailmne huvi kasv, mida illustreerib Google Trends'i [17] graafik joonisel 1. Need viimase nelja aasta (1. oktoober 2014 kuni 1. oktoober 2018) andmed peegeldavad ilmekalt AWS Lambda lansseerimist novembris 2014 ning „serverivaba“ termini hoovõttu aastal 2015. Jooniselt näeme veel, et „AWS Lambda“ on hetkel otsinguterminina populaarsem, kui üldisem ingliskeelne termin „serverless“, mis näitab

[1] https://www.youtube.com/watch?v=U8ODkSCJpJU
[2] http://serverlessconf.io

selle platvormi tuntust ja domineerimist võrreldes teiste teenusplatvormidega. Lisaks on märgata otsinguterminite vahel selget korreleerumist, mis suurendab andmete usaldusväärsust.



Joonis 1. „Serverless“ ja „AWS Lambda“ otsingute kasv aastatel 2014-2018 [17].

Järgnevates jaotistes defineeritakse täpsemalt „serverivaba“ olemust ning kirjeldatakse serverivaba andmetöötluse eeliseid ja puuduseid. Lisaks tuuakse välja ja võrreldakse peamiseid FaaS-teenusplatvorme.

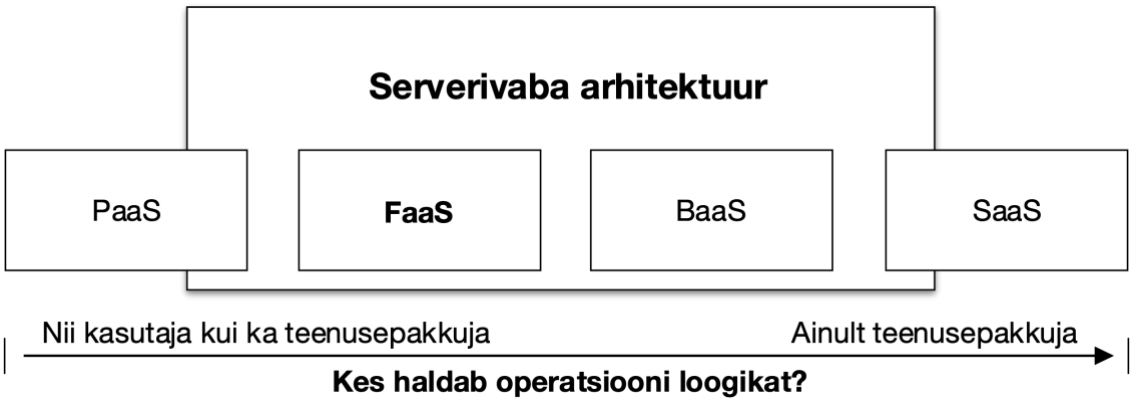
3.1 Mida tähendab serverivaba?

Sarnaselt teistele tehnoloogia trendidele puudub ühtne arvamus selle kohta, mis on „serverivaba“ [14], kuid peaaegu igaüks nõustub sellega, et kuigi kuskil on loomulikult olemas serverid, milles kood käivitub, on süsteem "serverivaba" siis, kui arendajad ei pea ise nende serverite pärast muretsema [18]. Aga kuidas sobitub serverivaba arhitektuur teiste NIST pilvandmetöötluse definitsiooni teenusmudelite kõrvale (nt PaaS, SaaS)?

Esmalt vaatleme, kuidas on NIST pilvandmetöötluse definitsioonis [19] sõnastatud PaaS ja SaaS. PaaS (ingl *Platform as a Service*) ehk platvorm teenusena on NIST definitsiooni kohaselt pilvandmetöötluse teenusmudel, milles tarbijale on antud võimalus juurutada enda loodud või omandatud rakendusi, mis on loodud programmeerimiskeeltest, teekidest, teenustest ja vahenditest, mida teenusepakkuja toetab. Tarbija ei halda ega kontrolli kasutusel olevat pilvinfrastruktuuri, sealhulgas võrku, servereid, operatsioonisüsteeme või mäluseadeid, kuid omab kontrolli rakenduste ja võimalike

keskkonna konfiguratsiooniseadete üle. SaaS (ingl *Software as a Service*) ehk tarkvara teenusena on NIST definitsiooni kohaselt pilvandmetöötluse teenusmudel, milles tarbijale pakutakse võimalust kasutada teenusepakkuja rakendusi pilvinfrastruktuuril. Rakendused on ligipääsetavad erinevatelt kliendi seadmetelt õhukese kliendiliidese kaudu, näiteks veebibrauseri või programmiliidese abil. Tarbija ei halda ega kontrolli kasutusel olevat pilvinfrastruktuuri, sealhulgas võrku, servereid, operatsioonisüsteeme, mäluseadeid või isegi üksikuid rakendusvõimalusi, välja arvatud rakenduste piiratud konfiguratsiooniseadeid.

Seif'i (SAP SE) ja Thömmes'i (IBM) artikli [20] põhjal peab serverivaba arhitektuuri paigutamiseks teiste pilvandmetöötluse teenusmudelite kõrvale vaatama, kes haldab pilveteenuse operatsioone. Vastust kujutab Joonis 2, millel on näha skaala operatsiooni loogika halduse järgi, alates nii kasutaja kui ka teenusepakkuja jagatud haldusest (PaaS) kuni ainult teenusepakkuja-poolse halduseni (SaaS). Sellest vaatevinklist lähtudes hõlmab serverivaba arhitektuur nii FaaS kui ka BaaS mudelit, mis paigutuvad PaaS ja SaaS mudelite vahele. Ühel pool paiknevat platvormi kui teenust (PaaS) ja teisel pool paiknevat tarkvara kui teenust (SaaS) üldjuhul serverivabaks ei peeta, kuid nad omavad serverivaba arhitektuuriga teatud ühiseid tunnuseid.



Joonis 2. Serverivaba arhitektuur ja pilvandmetöötluse teenusmudelid [20].

Eelkõige hõlmab serverivaba kahte erinevat, kuid kattuvat teenusmudelit: BaaS (ingl *Backend as a Service*) ja FaaS. Terminit „serverivaba" kasutati algselt selliste rakenduste kirjeldamiseks, mis kasutavad tagarakenduse loogika ja oleku haldamiseks täielikult või suurel osal kolmanda osapoole pilverakendusi ja teenuseid. Need on üldjuhul mobiiliäpid,

mis kasutavad autentimisteenuseid (nt Auth0[1]) ja pilvandmebaaside teenuseid (nt Firebase[2]). Selliseid teenuseid on varem kutsutud MBaaS (ingl *Mobile Backend as a Service*) või lihtsamalt BaaS [14].

Samas võib serverivaba tähendada ka rakendusi, kus arendaja loob endiselt tagarakenduse loogika, kuid erinevalt traditsioonilistest arhitektuuridest käivitatakse see mikroteenus sündmuselt juhitult olekuta konteinerites, mis eksisteerivad ainult lühikest aega ja mida haldab täielikult kolmas osapool. Üks võimalus seda kirjeldada on „funktsioon teenusena“ või lühemalt FaaS (ingl *Function as a Service*) [14]. Praegusel hetkel on kõige populaarsem FaaS-teenusplatvorm AWS Lambda, aga on ka mitmeid teisi (FaaS-platvormidest on täpsemalt juttu jaotises 3.4).

BaaS ja FaaS on oma omaduste poolest sarnased (nt puudub ressursside haldus) ja neid kasutatakse sageli koos [14]. Kõikidel pilveteenuse pakkujatel on omad "serverivaba andmetöötluse portfellid", mis sisaldavad nii BaaS'i kui ka FaaS'i tooteid – näiteks AWS'il on eraldi serverivabadele teenustele pühendatud veebileht[3]. Samuti omab Google Firebase BaaS andmebaasiteenus nüüd FaaS tuge läbi Google Cloud Functions teenusplatvormi [21].

PaaS'i (nt Google App Engine[4], AWS Elastic Beanstalk[5]) puhul ei pea kasutaja samuti serverite skaleerimist käsitsi ette võtma, kuid serverid on siiski nähtavad ning valima peab ressursid ja skaleerimise piirid. Teine oluline erinevus võrreldes FaaS'iga on see, et kui viimase puhul makstakse ainult reaalselt kasutatud arvutusressursi eest (üldiselt 100 ms kaupa), siis PaaS'i puhul peab alati vähemalt ühe serveri eest maksma, isegi kui seda ei kasutata. Teisest äärmusest leiame SaaS'i, mis on tarkvara teenusena müümise mudel lõpptarbijale (mitte arendajale), kellel pole ligipääsu ei serverile ega koodile – operatsiooni loogikat haldab ainult teenusepakkuja.

Viimasel ajal saab enim tähelepanu FaaS, seda peamiselt sellepärast, et see esindab serverivaba andmetöötluse juures kõige murrangulisemat aspekti ja muudab

[1] https://auth0.com
[2] https://firebase.google.com
[3] https://aws.amazon.com/serverless/
[4] https://cloud.google.com/appengine/
[5] https://aws.amazon.com/elasticbeanstalk/

fundamentaalselt seda, kuidas arendajad rakendusi loovad. Kuid FaaS on ainult üks osa serverivabast arhitektuurist [18]. Olemas on ka serverivabad andmesalvestusteenused, nagu Amazon S3[1] ja andmebaaside teenused (ingl *Database as a Service*, DBaaS), nagu Amazon DynamoDB[2]. Need on samuti serverivabad teenused selles mõttes, et kasutajal pole vaja seadistada sõltumatuid kettaid (RAID) või otsustada, kuidas andmebaasi klastrit tasakaalustada. Infrastruktuuri konfigureerimine, haldamine ja skaleerimine on kasutaja eest ära tehtud.

3.2 Serverivaba arhitektuuri eelised

Mis teeb serverivaba arhitektuuri eriliseks ja mis motiveerib seda kasutama? Järgnevalt kirjeldatakse täpsemalt serverivaba arhitektuuri peamisi eeliseid, lähtudes kasutaja perspektiivist ja keskendudes peamiselt FaaS'ile. Nendeks on suurem produktiivsus, automaatne skaleerimine ja kulude kokkuhoid ning lisaväärtusena pakub see „rohelisemat" andmetöötlust.

Suurem produktiivsus. Serverivaba arhitektuur võimaldab keskenduda koodile, samal ajal kui FaaS-teenusplatvorm hoolitseb kõige muu eest [22]. See vähendab oluliselt aega, mis kulub "ideest toodangukeskkonda" jõudmiseks. Lisaks sellele on serverivabasid süsteeme lihtsam disainida ja juurutada, kuna keerulised probleemid, nagu automaatne skaleerimine, kõrgkäideldavus ja serveri turvalisus, antakse FaaS-platvormi kanda. Suured pilveteenuse pakkujad võtavad enda peale raskemad probleemid ja mastaabisäästu tulemuseks on kulude kokkuhoid. Sel viisil saab arendaja keskenduda ainult koodile ja sellele, mida see tegema peab. Tõeline serverivaba lahendus ei vaja süsteemiadministreerimist.

Eesti idufirma Dashbird[3], mis pakub serverivabadele tarkvarasüsteemidele seiret ja tõrkeotsingut, viis oma klientide seas 2018. aasta aprillis läbi serverivaba teemalise uuringu [23]. Küsitlusele vastas 19 ettevõtet, kes kasutavad aktiivselt AWS Lambda't. Uuringu tulemustest selgus, et suurimaks serverivaba tehnoloogia eeliseks hinnati tarkvara arenduse lihtsamaks/kiiremaks tegemist (42%) ehk produktiivsust, mille kannul tulid automaatne skaleerimine (32%) ja kulude vähendamine (21%).

[1] https://aws.amazon.com/s3/
[2] https://aws.amazon.com/dynamodb/
[3] https://dashbird.io

Automaatne skaleerimine. Serverivabades lahendustes on automaatne skaleerimine sisse ehitatud, sest servereid haldavad kolmanda osapoole teenusepakkujad [24]. See tähendab, et alles jäävad aeg ja raha, mis tavaliselt kuluvad automaatse skaleerimise saavutamiseks. Lisaks skaleeruvusele suureneb ka käideldavus, kui pilveteenuse pakkujad haldavad oma arvutusvõimsust kättesaadavustsoonide ja regioonide üleselt, mis omakorda muudab serverivaba rakenduse turvaliseks ja kättesaadavaks, kuna see kaitseb seda piirkondlike rikete eest. Ressursihalduse eest hoolitseb pilveteenuse pakkuja – see hõlmab servereid, operatsioonisüsteeme, paikamisi, logimist, seiret, automaatset skaleerimist ja ülalhoidu.

Skaleeruvusega on seotud ka paralleelse töö võimekus. Teatud tüüpi rakendused võidavad oluliselt ajas, sest need saavad oma ülesandeid paralleelselt lahendada. See on üks oluline eelis, mida serverivaba arhitektuuriga rakendused omavad. Hea näide siin on automaattestid. Couch ja Waecher [25] näitasid oma uuringus, et tänu AWS Lambda teenuse kasutamisele vähendasid nad rakenduse Selenium baasil loodud kasutajaliidese testidele kuluvat aega varasemalt mitmelt tunnilt muljetavaldava 39 sekundini.

Kulude kokkuhoid. Serverivabade lahenduste puhul makstakse ainult päringute, mitte rakenduste olemasolu eest. See tähendab, et maksta tuleb täpselt selle eest, mida kasutatakse. Kuna rakenduse serverite haldust ja automaatset skaleerimist osutab pilveteenuse pakkuja ise, on need kulud samuti soodsamad võrreldes isetegemisega [24]. Enamik varem välja toodud juhtumiuuringutest (vt peatükk 2) märkis ära olulist pilveteenuste kulude kokkuhoidu [6]–[8]. Erandiks on sellist tüüpi rakendus, mis omab pidevat ja väga kõrget koormust [9].

Kui organisatsioon on juba otsustanud hallatava infrastruktuuriga arhitekuurid kõrvale jätta, kasutades ainult serverivabasid teenuseid, ei piirdu kulude kokkuhoid üksnes eelpool mainituga. Koos hallatava infrastruktuuriga kaob ära ka vajadus vastavate oskuste ja tööjõu järele [26]. Arvestades fakti, et spetsialiseeritud IT-tööjõud on üks kallimaid, toob see omadus endaga kaasa märkimisväärse kulude kokkuhoiu.

„Rohelisem“ andmetöötlus. Ajakirja Forbes andmetel [27] kasutavad tüüpilised serverid äri- ja suurettevõtete andmekeskustes keskmisel aasta jooksul vaid 5-15 protsenti nende maksimaalsest arvutusvõimsusest. See tähendab, et sisselülitatud tegevusetud serverid tarbivad märkimisväärsel hulgal energiat ja see on osalt põhjuseks, miks me

vajame nii palju ja järjest suuremaid andmekeskusi. See on erakordselt ebaefektiivne ja tekitab suure keskkonnamõju [14].

Kui kasutatakse majasiseseid servereid või IaaS ja PaaS infrastruktuuri lahendusi, peab tegema manuaalseid otsuseid rakenduse ressursi- ja jõudlusvajaduse osas ajaperioodideks, mis kestavad vahel kuid või aastaid [14]. Tavaliselt ollakse õigustatult ettevaatlikud ja määratakse serveritele liigselt ressurssi, mis põhjustabki serverite ressursi ebaefektiivse kasutamise.

Serverivaba lahenduse puhul ei tehta neid otsuseid enam ise, vaid lastakse pilveteenuse pakkujal eraldada täpselt nii palju andmetöötlusmahtu, kui sel hetkel tarvis läheb [14]. Teenusepakkujad saavad neid otsuseid ise teha, kombineerides kõikide klientide vajadusi. See erinevus toob kaasa andmekeskuste ressursside palju tõhusama kasutamise ja seega ka keskkonnamõjude vähendamise võrreldes traditsioonilise ressursihaldamisega.

3.3 Serverivaba arhitektuuri puudused

Enne uue tehnoloogia kasutuselevõttu tuleks kindlasti selgusele jõuda selle puudustes. Serverivaba arhitektuuri puudused saab jagada kaheks. Esimest tüüpi puudused on serverivabale kontseptsioonile omased – nendega peab alati arvestama [14]. Teist tüüpi puudused on seotud praeguste FaaS-teenusplatvormide (AWS Lambda jt) tehnilise rakendusega – aja möödudes need tõenäoliselt kõrvaldatakse.

Esmalt kirjeldame serverivaba kontseptsiooni iseärasusi, millest peamised on tootjalukustus (ingl *vendor lock-in*), turvalisusküsimused ja see, et FaaS-funktsioonide olek ei säilu. Tehniliste puuduste osas toome välja kaks peamist, milleks on külmkäivitus (ingl *cold start*) ning piiratud käitusaeg ja mälumaht.

Tootjalukustus. Ehkki väga väike osa käivitatavast koodist sõltub FaaS-platvormist endast (platvorm ainult käivitab konkreetse funktsiooni), võib serverivabas keskkonnas töötamine muuta rakenduse koodi sõltuvaks pilveteenuse pakkuja kümnetest teistest teenustest [6]. Kuigi Serverless[1] raamistik ja teised sarnased tööriistad võivad lihtsustada koodi juurutamist erinevate teenusplatvormide vahel, on serverivabad rakendused suure tõenäosusega tihedalt seotud ühe kindla pilveteenuse pakkuja teenustega. Praktikas

[1] https://serverlesss.com

tähendab see seda, et rakenduse üleviimine ühelt pilveteenuse pakkujalt teisele nõuab olulisel määral koodi ümberkirjutamist.

Turvalisusküsimused. Serverivaba lähenemisviisi kasutuselevõtmine toob kaasa hulga turvalisusküsimusi. Iga uue serverivaba teenuse kasutamine toob juurde erinevaid turvalisuse rakendusviise, ala pahatahtlike kavatsuste teostamiseks suureneb ja tõuseb eduka ründe tõenäosus. Võttes kasutusele FaaS'i, tekib märkamatult suur hulk eraldiseisvaid funktsioone, millest igaüks on uus ohuvektor [14]. Oluline on erinevaid riske teada ja minimeerida.

Ettevõte PureSec[1], mis pakub platvormi serverivabade arhitektuuride turvalisemaks muutmiseks, on avaldanud põhjaliku uurimuse [28] kümnest peamisest turvalisuse riskist, mis on serverivabadele arhitektuuridele omased. Uurimusest järeldub, et serverivabade arhitektuuride arendamisel on hulk turvalisuse riske, mida peab teadma ja silmas pidama. Need on (alates kõige kriitilisemast):

- Funktsiooni sündmuse andmete süst (ingl *Function Event-Data Injection*)
- Vigane autentimine (ingl *Broken Authentication*)
- Ebaturvaline juurutamise konfiguratsioon (ingl *Insecure Serverless Deployment Configuration*)
- Üleprivilegeeritud funktsiooni õigused ja rollid (ingl *Over-Privileged Function Permissions and Roles*)
- Ebapiisav funktsioonide seire ja logimine (ingl *Inadequate Function Monitoring and Logging*)
- Ebaturvalised kolmanda osapoole sõltuvused (ingl *Insecure 3rd Party Dependencies*)
- Ebaturvaline rakenduse saladuste hoiustamine (ingl *Insecure Application Secrets Storage*)
- Teenusetõkestamine ja finantsressursside ammendumine (ingl *Denial of Service & Financial Resource Exhaustion*)
- Funktsioonide täitmise voo manipuleerimine (ingl *Functions Execution Flow Manipulation*)

[1] https://www.puresec.io

- Väär eranditöötlus ja paljusõnalised veateated (ingl *Improper Exception Handling and Verbose Error Messages*)

FaaS-funktsioonide olek ei säilu. Funktsioonide hoidmine olekuta võimaldab FaaS-teenusplatvormil käivitada vajalikul hulgal uusi funktsioonide konteinereid, et skaleerida vastavalt sissetulevate päringute arvule [29]. FaaS-programmeerimismudel on olekuta, aga kood omab olekule ligipääsu selliste teenuste abil, nagu näiteks Amazon S3 andmetalletus või Amazon DynamoDB andmebaas. Kuigi FaaS-funktsioonide konteinerite täpset eluiga ei saa määrata, on eksperimentide [30] tulemusel leitud, et näiteks AWS Lambda puhul hävitatakse tegevusetud konteinerid üldjuhul alles 45-60 minuti jooksul pärast viimast päringut. Samas vahel võidakse need ka palju varem hävitada, et vabastada ressursse teiste klientide jaoks.

Võttes arvesse, et varem loodud konteinereid võidakse uute päringute puhul taaskasutada, on AWS Lambda parimates praktikates soovitatud kirjutada kood nii, et sellel oleks võimekus teatud olekuid uuesti kasutada – näiteks andmebaasi ühendusi, mis loodi varasema päringu jooksul [31].

Külmkäivitus. See on latentsusaeg, mis lisandub, kui FaaS-funktsiooni koodi käivitamiseks pole saadaval ühtegi varasemalt loodud konteinerit [32]. See on platvormi kasutajatele nähtamatu ja teenusepakkuja käes on täielik kontroll konteinerite hävitamise aja osas.

Külmkäivituse probleemi osas tuleb veel ära märkida, et seda mõjutab nii käitusmootor (sõltuvalt sellest, millises programmeerimiskeeles funktsioon kirjutatud on) kui ka määratud põhimälu suurus. C# ja Java puhul on külmkäivituse latentsusaeg palju suurem kui Python ja Node.js puhul. Lisaks suurendab mälumaht külmkäivitust lineaarselt (mida rohkem mälu kasutatakse, seda kauem kulub selle käivitamiseks) [24], [33].

Viimaks peab FaaS-funktsioonide loomisel ja seadistamisel meeles pidama, et kui näiteks AWS Lambda't kasutades asetada funktsioon VPC (ingl *Virtual Private Cloud*) sisse, mis on üldiselt hea tava, tuleb arvestada oluliselt pikema latentsusajaga, mis võib ulatuda kuni 10 sekundini [34].

Piiratud käitusaeg ja mälumaht. FaaS-funktsioonidel on maksimaalse seadistatava käitusaja ja mälumahu piirang. Näiteks käesoleva töö kirjutamise hetkel on AWS Lambda

puhul maksimaalne käitusaeg 15 minutit ja mälumaht 3008 MB [35]. Rakenduse arendajatel pole võimalust neid piiranguid ületada, käitusaja puhul peavad kõik üksikud ülesanded selle ajavahemiku jooksul lõpule jõudma. AWS pakub mitmeid töövoogude lahendusi, et siduda ja ühendada Lambda funktsioone pikemaks jadaks, kuid see siiski ei eemalda üksiku ülesande käitusaja piirangut [6]. See tähendab, et FaaS ei ole optimaalne lahendus sügavõppe rakendustele, või millelegi muule, mis vajab suuri mälumahte või on mõeldud pikka aega töötamiseks [36].

3.4 FaaS-platvormid

Serverivaba arhitektuuriga rakenduse võtmekomponendiks on FaaS-teenusplatvorm, kuhu kood üles laetakse ja milles serverite ülalpidamise ja skaleerimise eest vastutab pilveteenuse pakkuja. Lisaks tuntud teenusplatvormidele on avalikustatud ka hulk avatud lähtekoodiga platvorme, mis populariseerivad ja laiendavad FaaS-kontseptsiooni ning loovad aluse uute teenusplatvormide loomisele. Selleks, et teha platvormi valiku osas parem ja kaalutletum otsus, võrreldakse käesolevas jaotises peamiseid hetkel saadaval olevaid teenusplatvorme ja antakse ülevaade avatud lähtekoodiga projektidest.

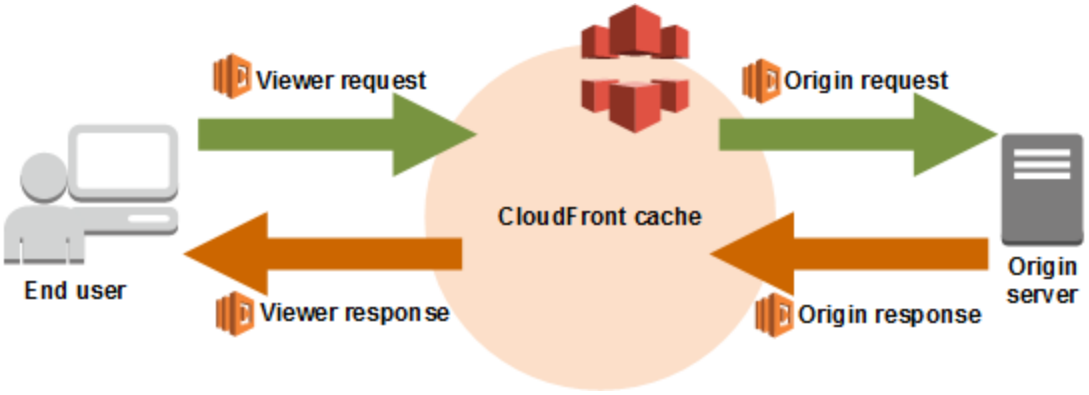
3.4.1 Teenusplatvormid

Esimene samm serverivaba arhitektuuri kasutuselevõtuks on tutvumine peamiste FaaS-teenusplatvormide ja nende omadustega, et valida endale sobivaim. Tabel 1 esitab nende põhjaliku võrdluse (seisuga 06.05.2019). Võrdluskriteeriumideks on lähtekood (avatud või kinnine), teenusetaseme leping (ingl *Service Level Agreement*, SLA), toetatud keeled (programeerimiskeeled, milles saab funktsioone arendada), mälumaht (kui palju mälu on võimalik funktsioonidele eraldada), maksimaalne koodi maht, maksimaalne käitusaeg ja maksimaalne samaaegsete päringute arv. Viimane oluline võrdluskriteerium on esitatud Lisas 1, milleks on FaaS-teenusplatvormide sisseehitatud päästikud (ingl *triggers*) – need on erinevad viisid, kuidas FaaS-funktsioone vastavas teenusplatvormis käivitada. Neli tuntumat FaaS-teenusplatvormi, mida antud peatükis järgnevalt võrreldakse, pärinevad suurimatelt pilveteenuse pakkujatelt. Need on Amazon'i AWS Lambda, Microsoft'i Azure Functions, Google Cloud Functions ja IBM Cloud Functions.

Serverivaba andmetöötluse teerajajaks peetakse **AWS Lambda**'t, mis näib olevat hetkel isegi tuntum kui „serverivaba" märksõna ise (vt Joonis 1). See avalikustati 2014. aasta novembris [37] ja 3 aastat hiljem tagati ka 99,95% teenusetaseme lepinguga [38]. Seega

on Lambda tehnoloogia piisavalt arenenud ja stabiilne ning valmis kasutamiseks toodangukeskkondades. Lambda’t saab kasutada selleks, et kood reageeriks sündmustele, näiteks muudatustele Amazon S3 failikataloogis või Amazon DynamoDB andmebaasi tabelis, käivitades koodi vastusena HTTP-päringutele, kasutades Amazon API Gateway teenust või käivitades koodi AWS SDK abil tehtud API-päringute läbi. Nende võimaluste abil saab luua serverivaba andmetöötluse töövoogusid või tagarakendusi, mis on samaväärsed AWS’i enda skaleerimise, jõudluse ja turvalisusega [39]. Lisaks pakub AWS võimalust käivitada Lambda funktsioone lõppkasutajale lähimas AWS’i andmekeskuses, et kohandada Amazon CloudFront'i sisuedastusvõrgus pakutavat sisu. Selle teenuse nimetuseks on Lambda@Edge. Funktsioonid käivituvad vastuseks CloudFront’i sündmustele, ilma serverite hooldamise või haldamiseta. Lambda funktsioonide abil saab muuta CloudFront'i päringuid ja vastuseid järgmistes punktides (vt Joonis 3) [40]:

- Kui CloudFront saab vaatajalt päringu (ingl *viewer request*)
- Enne, kui CloudFront edastab päringu allikasse (ingl *origin request*)
- Kui CloudFront saab vastuse allikast (ingl *origin response*)
- Enne, kui CloudFront saadab vastuse vaatajale (ingl *viewer response*)



Joonis 3. Lambda@Edge sündmused Amazon CloudFront sisuedastusvõrgus [40].

AWS Lambda on võrreldes teiste teenusplatvormidega kõige vanem ja arenenum ning omab üldiselt ka kõige suuremaid maksimaalseid piiranguid (mälu, käitusaeg, samaaegsete päringute arv). Samuti on tal kõige rohkem sisseehitatud päästikuid (vt Lisa 1), kuna AWS enda teenuste portfell on kõigi pilveteenuse pakkujate hulgas suurim. Lisaks suurele hulgale toetatud keeltele pakub Lambda nüüd võimalust läbi Lambda Runtime API lisada tuge ükskõik millise programmeerimiskeele jaoks [41].

Microsoft Azure avalikustas **Azure Functions**'i esmakordselt aastal 2016 [42]. Erinevalt teistest teenusepakkujatest pakub Azure oma FaaS-teenusplatvormi kasutamiseks kahte erinevat paketti. Esimene, „Consumption" pakett, on tüüpiline serverivaba teenus, mille omadusteks on automaatne skaleerimine ja päringute põhine hinnastamine, mis teevad selle võrdväärseks teiste teenusplatvormidega. Samas teine, „App Service" pakett, on oma olemuselt pigem sarnasem PaaS'ile – sellel on erinevad ressursside jaotamise tasemed, mille vahel kasutaja peab valima. See tähendab, et kasutaja põhimõtteliselt reserveerib ja hoiab seeläbi „soojas" soovitud arvu konteinereid iga funktsiooni jaoks. „App Service" pakett pakub rohkem võimalusi ja suuremaid piiranguid, kuid on seotud püsikuludega. Seega pole tegemist selles mõttes serverivaba teenusega, et ikkagi peab serverite peale mõtlema ja planeerima nende võimalikku koormust [43]. Mõlemad paketid on tagatud 99,95% teenusetaseme lepinguga. 2019. aasta aprillis teatas Microsoft [44], et on tulemas uus ettevõtetele mõeldud „Premium" pakett, mis paigutub omadustelt kahe olemasoleva vahele.

Kõige uuem suurte pilveteenuse pakkujate FaaS-teenusplatvormidest on **Google Cloud Functions**, mis avalikustati esmakordselt aastal 2017 [45] ning väljus *beta*-staatusest 2018. aasta augustis [46]. Kuivõrd see on kõige uuem, on sellel hetkel ka kõige vähem võimalusi. Näiteks programmeerimiskeeltest on hetkel toetatud ainult Node.js, Python ja Go ning maksimaalne limiit käitusajale on kõige madalam. Samuti on ette näidata kõige vähem sisseehitatud päästikuid (vt Lisa 1). Kuid sellele vaatamata on kvaliteet tagatud 99,5% teenustaseme lepinguga ning juba lähiajal on teadaolevalt tulemas palju uuendusi, kaasa arvatud serverivabade konteinerite formaadi (Dockerfile) võimekus [46].

IBM Cloud Functions avalikustati 2016. aasta veebruaris [47]. See oli esimene suure pilveteenuse pakkuja FaaS-teenusplatvorm, mis avalikustati avatud lähtekoodiga. Üldiste omaduste poolest on see võrreldav AWS'i ja Azure toodetega, sh samaväärse 99,95% teenusetaseme lepinguga. Samas pakub see Dockerfile baasil funktsioonide loomise tuge, mida teised kolm hetkel veel ei paku (kuigi Google Cloud Functions tegevuskavas on see juba sees). Ühtlasi on IBM Cloud Functions võrreldes teistega üks hinna poolest soodsamaid (vt Tabel 2).

Tabel 1. Peamiste FaaS-teenusplatvormide omaduste võrdlus.

	AWS Lambda	**Azure Functions**	**Google Cloud Functions**	**IBM Cloud Functions**
Pilveteenuse pakkuja	Amazon Web Services	Microsoft Azure	Google Cloud Platform	IBM Cloud
Lähtekood	Kinnine	Avatud[1]	Kinnine	Avatud (Apache OpenWhisk)
Teenusetaseme leping (SLA)	99,95%	99,95%	99,5%	99,95%
Toetatud keeled	Node.js (JavaScript) Java Python C# Go Ruby .NET kohandatav	Node.js (JavaScript) Java C# F# Powershell	Node.js (JavaScript) Python Go	Node.js (JavaScript) Java Python Go Swift PHP .NET Ruby Docker
Mälumaht	128 MB - 3008 MB	128 MB - 1536 MB	128 MB - 2048 MB	128 MB - 2048 MB
Maksimaalne koodi maht	50 MB (kokkupakituna) 250 MB (lahtipakituna)	Limiit puudub aga failide hoiustamise eest peab eraldi maksma	100 MB (kokkupakituna) 500 MB (lahtipakituna)	48 MB
Maksimaalne käitusaeg	15 minutit	„Consumption“ pakett: 10 minutit „App Service“ pakett: piiranguta	9 minutit	10 minutit
Maksimaalne samaaegsete päringute arv	Vaikimisi 1000 konto kohta aga saab paluda suurendamist	Piirang on täpsustamata	100 000 000 iga 100 sekundi kohta (regiooni piirang)	Vaikimisi 1000 konto kohta, aga saab paluda suurendamist

[1] https://github.com/Azure/azure-functions-host/

Platvormide valiku ja võrdluse puhul on oluline ka nende hinnastamist võrrelda. Tabel 2 esitab peamiste FaaS-teenusplatvormide hinnastamise võrdluse (seisuga 06.05.2019). Võrdluskriteeriumideks sai valitud käitusaja ümardamise loogika, hinnakomponendid (käivitused, RAM, CPU, võrguliiklus, API-päringud) ning viimaks erinevate teenusplatvormide kuu jooksul tasuta pakutavad ressursid. Sellest on näha, et igal teenusepakkujal on omanäoline hinnastamise mudel ja reeglid. Ent kõikide iseärasuste kõrval on kaks põhimõtet samad: iga käitusaja eest kuulub tasumisele vähemalt 100 ms ja igal kuul on teatud osa teenusest tasuta.

Tuleb tunnistada, et FaaS'i hinnastamine on keeruline. Lisaks otsestele käivituste ning CPU ja RAM kuludele peab arvestama ka võrguliikluse ja API-päringutega. Viimane neist võib põhjustada lõviosa kuludest. API-päringute funktsionaalsus on vajalik, kui on soov siduda FaaS-funktsioonid ühtseks API'ks, millel on muutumatu veebiaadress. AWS pakub seda võimalust läbi enda API Gateway[1] teenuse, mis samas sisaldab endas palju enamat, kui muutumatute veebiaadresside loomist. Kuid kõnealust baasvõimalust pakuvad Azure ja IBM Cloud tasuta, mis võib tekitada suure kulude erinevuse teenusepakkujate vahel.

2018. aasta novembris, 4 aastat pärast AWS Lambda avalikustamist, kuulutas AWS oma iga-aastasel re:Invent konverentsil välja võimaluse kasutada Elastic Load Balancing (ELB) toote Application Load Balancer (ALB) tüüpi koormuse tasakaalustajat koos AWS Lambda funktsioonidega. See oli oluline uudis, sest enne seda oli AWS'i ainus ametlik viis luua enda domeeniga (nt api.surveer.com) Lamba funktsioonidel põhinevat REST API't üksnes Amazon API Gateway teenusega. Samas on Lambda kasutamisel ALB'i (võrreldes API Gateway'ga) üks põhimõtteline erinevus – ALB nõuab püsitasu isegi siis, kui seotud Lambda funktsioonid ühtegi korda ei käivitu. Näiteks AWS Stockholmi (eu-north-1) regioonis maksab üks ALB koormuse tasakaalustaja 0,02394 USD tunnis, mis teeb kuutasuks 17,5 USD [48]. Sellele hinnale lisandub veel tarbimispõhine LCU (Load Balancer Capacity Unit) lisatasu, mis on ühe ühiku kohta 0,0076 USD tunnis. Kuna ALB'i hinnastamine ei ole ainult tarbimispõhine, siis Tabel 2 seda ei esita. Kuid sellegipoolest tõotab ALB'i kasutamine olla suuremate koormuste puhul API Gateway kõrval oluliselt soodsam. Täpsemalt uurib käesolev töö seda kuluanalüüsis jaotises 5.2.

[1] https://aws.amazon.com/api-gateway/

Tabel 2. Peamiste FaaS-teenusplatvormide hinnastamise võrdlus.

	AWS Lambda	**Azure Functions (Consumption)**	**Google Cloud Functions**	**IBM Cloud Functions**
Käitusaja ümardamine (üles)	100 ms	1 ms (pärast esimest 100 ms)	100 ms	100 ms
Käivituste hind (1M kohta)	$0,21	$0,20	$0,40	-
Aeg/RAM hind (1M GB-sek)	$16,67	$16,00	$2,50	$17,00
Aeg/CPU hind (1M GHz-sek)	-	-	$10,00	-
Võrguliikluse hind (GB kohta)	$0,09	$0,087	$0,12	$0,09
API-päringute hind (1M kohta)	$3,50 (API Gateway)	Tasuta	$3,00	Tasuta
Tasuta käivitusi (kuu jooksul)	1M	1M	2M	-
Tasuta aeg/RAM (kuu jooksul)	400K	400K	400K	400K
Tasuta aeg/CPU (kuu jooksul)	-	-	200K	-
Tasuta võrguliiklus (kuu jooksul)	1 GB (EC2 üleselt)	5 GB (kõige üleselt)	5 GB (spetsiaalselt)	Täpsustamata

Lisaks tabelites võrreldavale väärib väljatoomist ka kaks uut ja omanäolist alternatiivi, mida Serverless raamistik on toetama asunud. Esimene, Spotinst Functions[1], reklaamib ennast soodsaima FaaS-teenusplatvormina, mis erinevalt eelmainitutest ei oma ise andmekeskusi, vaid kasutab erinevate pilveteenuse pakkujate allahinnatud *spot*-tüüpi pilveservereid [49]. Teine, Cloudflare Workers[2], pakub seevastu sarnaselt AWS Lambda@Edge'ile lihtsate FaaS-funktsioonide loomist enda ülemaailmses sisuedastusvõrgus, mille kood käivitub lõppkasutajale lähimas andmekeskuses. Nende eriline lähenemine ei kasuta konteinerite tehnoloogiat, vaid V8 JavaScript mootorit, hoidudes seeläbi konteinerite külmkäivituse ja latentsusaja probleemidest [50].

[1] https://spotinst.com/products/spotinst-functions/

[2] https://www.cloudflare.com/products/cloudflare-workers/

3.4.2 Avatud lähtekoodiga platvormid

Pilveteenuse pakkujate (suuresti kinnise lähtekoodiga) FaaS-teenusplatvormide kõrvale on ilmunud mitmed avatud lähtekoodiga platvormid, mis võimaldavad soovijatel ise FaaS-platvorme juurutada ja opereerida vabalt valitud infrastruktuuril.

Võrreldes FaaS-platvormi kasutajatega on platvormi opereerijatel erinevad, kuid seotud eelised. Peamine idee on sama – ressursside väga tõhus kasutamine. Erinevalt rakendusest, mis kasutab ressursse 24 tundi ööpäevas (olenemata sellest, kas see on kasutusel või mitte), on FaaS-funktsioonid kogu infrastruktuuri lõikes jaotatud ja tarbivad ressursse ainult siis, kui nad tegelikult midagi teevad. Samuti on juurutatud FaaS-platvormi lihtne hallata ja skaleerida, sest skaleerimine toimib iga funktsiooni jaoks ühesuguselt ning jälgima peab ühtset süsteemi [51].

Tabel 3 esitab tuntumad avatud lähtekoodiga FaaS-platvormid (seisuga 03.05.2019). Võrdluskriteeriumideks on platvormi programmeerimiskeel (mis keeles see kirjutatud on), toetatud keeled (mis keeltes saab funktsioone kirjutada) ja GitHub'i projekti tähtede arv ümardatult. Viimase järgi näeme projektide suhtelist populaarsust, mis ei ole teaduslik hinnang, kuid annab praktilise võrdluspildi. Selle alusel paistab välja OpenFaaS, mis on kõikidest teistest umbes 3 korda populaarsem.

Keelte osas paistab silma, et Go programmeerimiskeeles on kirjutatud kõik platvormid peale Apache OpenWhisk'i, mis on kirjutatud Scala's. Stack Overflow 2018. aasta uuringu põhjal on professionaalsete arendajate seas Go pisut populaarsem kui Scala (7,2% vs 4,5%) [52]. See annab Go keeles kirjutatud projektidele ehk väikese eelise suurema jätkusuutlikkuse näol.

Toetatud keelte osas tuleb projektide vahel välja suurem põhimõttelisem erinevus. Tõsi, kõik FaaS-platvormid põhinevad konteinerite tehnoloogial, kuid sarnaselt peamistele teenusplatvormidele toetab enamik funktsioonide kirjutamist ainult kindlates programmeerimiskeeltes. Paljud projektid pakuvad kohandamise võimalust, et toetada uusi keeli, kuid OpenFaaS on põhimõtteliselt erinev. Docker'i kasutajatele tuttavalt defineeritakse kõik konteinerid Dockerfile'i abil. Käsurea liidese abil on võimalik küll kasutada malle konkreetsete keelte jaoks, kuid üldiselt on oodatav sisend Dockerfile formaadis. See idee näib olevat veenev, sest lisaks Google Cloud'ile on ka näiteks avatud

lähtekoodiga pilvetoodete ja -teenuste ettevõte ZEIT arendamas [53] sarnast Dockerfile'il põhinevat serverivaba teenusplatvormi.

Tabel 3. Avatud lähtekoodiga FaaS-platvormide võrdlus.

Platvorm	**Keel**	**Toetatud keeled**	**GitHub'i tähti**
Apache OpenWhisk[1]	Scala	Node.js, Python, Go, Ruby, Java, Swift, Scala, PHP, Ballerina ja kohandatav	3980
Kubeless[2]	Go	Node.js, Python, Go, Ruby, Java, PHP, .NET, Ballerina ja kohandatav	4579
Fn[3]	Go	Node.js, Python, Go, Ruby, Java, Kotlin ja Docker image	3978
OpenFaaS[4]	Go	**Dockerfile** ja populaarsete keelte mallid	**14 073**
Fission[5]	Go	Node.js, Python, Go, Ruby, JVM, C#, PHP, .NET, Perl ja kohandatav	4292
Nuclio[6]	Go	Node.js, Python, Go, Java, .NET ja shell	2696
IronFunctions[7]	Go	AWS Lambda formaat ja kohandatav	2578

Kui Apache OpenWhisk kõrvale jätta, siis kõikidel teistel avatud lähekoodiga FaaS-platvormidel puuduvad seotud pilveteenuse pakkujad. Kõiki neid platvorme on võimalik ise vabalt valitud pilves või serverites konteinerite tehnoloogia (nt Docker, Kubernetes) abil juurutada.

Pole teada, kas sarnaselt IBM Cloud'ile ja Microsoft Azure'ile plaanib veel mõni suur pilveteenuse pakkuja oma FaaS-platvormi lähtekoodi avalikuks teha, kuid see on igal juhul tervitatav eeskuju. Samuti pole teada, millised avatud lähtekoodiga projektid alles jäävad, kas OpenFaaS kasvab veelgi populaarsuses või saab standardiks mõni uus projekt, mida pole veel arendama asutudki. Ent selge on see, et iga avatud lähtekoodiga FaaS-platvorm toob rohkem tähelepanu serverivabadele arhitektuuridele ning viib valdkonda edasi. Lisaks rajavad need projektid aluse uute FaaS-teenusplatvormide loomiseks.

[1] https://github.com/apache/incubator-openwhisk
[2] https://github.com/kubeless/kubeless
[3] https://github.com/fnproject/fn
[4] https://github.com/openfaas/faas
[5] https://github.com/fission/fission
[6] https://github.com/nuclio/nuclio
[7] https://github.com/iron-io/functions

Teostatud FaaS-platvormide võrdlus on aluseks töö autorile otsuse tegemisel, milline platvorm võtta Surveeri uue serverivaba arhitektuuri puhul kasutusele (vt jaotis 4.3). Kuigi praktilisemat abi pakkus teenusplatvormide võrdlus, andis avatud lähtekoodiga FaaS-platvormi ülevaade ideid kas ja kuhu oleks võimalik rakendusega edasi liikuda (migreerida), kui tekib mistahes põhjusel vajadus suurema (nt riistvara) kontrolli järele.

4 Küsitlustarkvara Surveer

Surveer on autori bakalaureusetöö raames arendatud veebipõhine tarkvara küsitluste ja uuringute läbiviimiseks. Olulisteks omadusteks on kaasaegne disain, kasutajasõbralik kasutajaliides ja mobiilisõbralik rakendus. Küsitlustarkvara pakub klientidele olulist väärtust – lihtsustab ankeetide koostamist ja andmete elektroonset kogumist. Lisaks pakub lihtsaid võimalusi andmete analüüsimiseks, eksportimiseks ja jagamiseks. Pärast küsitluste läbiviimist sisaldub klientide jaoks kõige olulisem väärtus kogutud andmetes.

Surveeri tarkvara pakutakse teenusena (SaaS). Ärimudeliks on nn *freemium* – klientidele pakutakse piiratud võimalustega tasuta paketti ja funktsionaalsemaid tasulisi pakette. Mudeli eelis võrreldes ainult tasuliste pakettidega on jõuda suurema hulga inimesteni ja pöörata potentsiaalsed tasuta paketi kasutajad tasulisteks klientideks. Maksmine toimub elektroonsete maksetena tellimuse põhiselt kuu või aasta kaupa.

Toote potentsiaalsed kliendid on kõik, kes soovivad ise küsitlusi läbi viia. Esmane sihtturg on Eesti, seejärel naaberriigid ja USA. Kaks olulisemat kliendigruppi on mikro- ja väikeettevõtted, kes soovivad teha personali rahuolu-, kliendirahulolu- ja/või turu-uuringuid ning üliõpilased ja teadurid, kes viivad läbi akadeemilisi uuringuid.

Järgnevates jaotistes kirjeldatakse Surveeri varasemat arhitektuuri, sellega esile kerkinud probleeme ning viimaks uue serverivaba arhitektuuri kasutuselevõttu, mis peaks nendele probleemide leevendust pakkuma.

4.1 Surveeri varasem arhitektuur

Selleks, et paremini mõista käesolevas töös kirjeldamisele tulevat migratsiooni serverivabale arhitektuurile, selgitatakse antud peatükis Surveeri varasemat (migratsioonile eelnevat) arhitektuuri. Lisaks on see oluline, et näha ilmnenud probleeme, mis motiveerisid migratsiooni ette võtma.

Surveer loodi MEAN-platvormil, mille akronüüm viitab MongoDB, Express, AngularJS ja Node.js üheskoos kasutamisele. Sellises arhitektuuris on ees- ja tagarakendused loodud

teineteisest sõltumatute komponentidena, kuid mõlemas pooles on kasutusel sama programmeerimiskeel – JavaScript.

Ülevaate Surveeri varasemast arhitektuurist annab Joonis 4, kus on näha rakenduse erinevaid komponente ja andmete liikumist erinevate kasutusel olevate (peamiselt AWS) pilveteenuste vahel. Üldiselt ülesehituselt koosneb Surveeri varasem arhitektuur kolmest eraldiseisva koodibaasiga alamrakendusest, mis on kõik koos juurdomeenil surveer.com:

1. **App** – AngularJS eesrakendus küsitluste loomiseks ja tulemuste analüüsimiseks, mis sisaldab endas ka avalikkusele suunatud staatilist kodulehte.
2. **Collector** – AngularJS eesrakendus küsitlusele vastamiseks.
3. **API** – monoliitne Node.js tagarakendus REST-otspunktidega, millega eesrakendused suhtlevad.

DNS-teenusena on kasutusel Amazon Route 53[1]. See on skaleeruv ja kõrgkäideldavusega tagatud teenus, et ühendada kasutajate päringud erinevate kasutusel olevate AWS'i infrastruktuuri teenustega, nagu Elastic Load Balancing[2] (ELB) koormuse tasakaalustaja ja Amazon Elastic Compute Cloud[3] (EC2) pilveserverid.

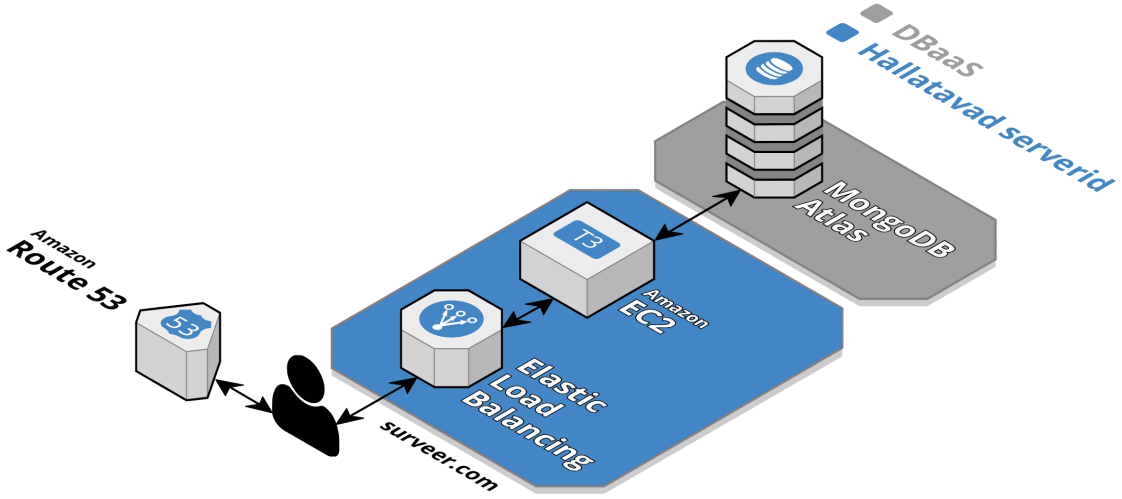
Kliendi poolel (veebibrauseris) käivitatakse AngularJS eesrakendused (App ja Collector), mis saadetakse sinna Nginx veebiserverist staatiliste (HTML, CSS, JavaScript jm) failidena. Andmete liikumine käib krüpteeritud kujul üle HTTPS protokolli. Eesrakendused pärivad andmeid Ajax-päringutega läbi Nginx veebiserveri, mis toimib sel juhul pöördproksina. Päringutele annab JSON vastused Node.js tagarakendus (API).

Nginx veebiserver, koos eesrakenduste (App ja Collector) staatiliste failide ja Node.js tagarakendusega (API), on juurutatud ning hallatud läbi AWS Elastic Beanstalk[4] PaaS teenuse, mis lihtsustab erinevate AWS pilveteenuste (nt ELB, EC2) koos kasutamist, automaatset skaleerimist ja mitmete sama arhitektuuriga keskkondade loomist.

Andmeid hoiustatakse MongoDB andmebaasis, mis on kasutusel läbi MongoDB Atlas[5] andmebaasiteenuse (DBaaS). Andmebaasi klaster, mis koosneb kolmest pilveserverist,

[1] https://aws.amazon.com/route53/
[2] https://aws.amazon.com/elasticloadbalancing/
[3] https://aws.amazon.com/ec2/
[4] https://aws.amazon.com/elasticbeanstalk/
[5] https://www.mongodb.com/cloud/atlas

on juurutatud API tagarakenduse EC2 pilveserveritega samasse AWS andmekeskusesse, et saavutada võimalikult lühikest latentsusaega ja on seadistatud suurema turvalisuse saavutamiseks samasse VPC-võrku, et piirata andmebaasile Internetist otsest ligipääsu.



Joonis 4. Surveeri varasem arhitektuur.

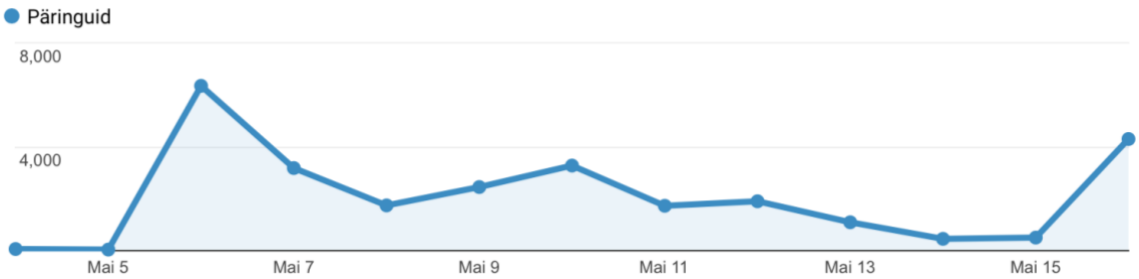
Surveer on juurutatud kolme sarnase arhitektuuriga keskkonda, milleks on: arenduskeskkond (dev), testkeskkond (staging) ja toodangukeskkond (prod). Arenduskeskkond on mõeldud tarkvara uuenduste testimiseks eraldiseisva andmebaasiga. Testkeskkond on mõeldud tarkvara uuenduste testimiseks võrdväärse infrastruktuuri ja toodangukeskkonna andmebaasiga ning toodangukeskkond on avalikkusele kasutamiseks. Arendus- ja testkeskkond on piiratud ligipääsuga.

4.2 Surveeri varasema arhitektuuri puudused

Olgugi, et Surveeri varasema, MEAN-platvormil põhineva arhitektuuriga, ei tulnud ette märkimisväärseid teenusetõrkeid või lahendamatuid skaleeruvuse probleeme, suurenesid kasutajatebaasi kasvades riskid ja vajadus pideva serverite haldamise järele. Kuna Surveeri visioon on võimalikult kaua tegutseda mikroettevõttena, siis sellest lähtuvalt on aktuaalne ka tööde ja töökohtade arvu minimeerimine. Üks sellistest rollidest on ööpäevaringne süsteemiadministraator. Serverivaba arhitektuuri kasutuselevõtt vähendaks vajadust sellise rolli järele, kuna kaovad ära administreerimist vajavad serverid.

Kõige ärikriitilisem osa veebipõhisest küsitlustarkvarast on küsitlustele vastamise rakendus. See on klientide jaoks kõige olulisem funktsionaalsus – andmete kogumine. Kui teenusetõrke tagajärjel ei ole võimalik küsitluse veebiankeeti avada või sellele vastata, võivad kliendid andmetest ilma jääda. See võib omakorda tekitada ettevõtte mainekahju ning põhjustada olemasolevate ja potentsiaalsete klientide kaotuse. Küsitluste loomise ja analüüsimise rakendus on samuti kasutajatele tähtis, kuid võimaliku andmekao negatiivne tagajärg oleks küsitlustele vastamise puhul suurusjärgus 10 kuni 100 korda suurem, sest küsitlustel on keskmiselt just nii palju vastajaid.

Erinevalt küsitluse loomise rakendusest (mis vastab üldjoontes korrapärasele mustrile) on küsitlustele vastamist keeruline ette ennustada ning see kasvab ja kahaneb hüppeliselt. Seda iseärasust illustreerib Joonis 5, millel on kujutatud Google Analytics'i graafik Surveeri küsitluste vastamise statistika kohta kahenädalasel perioodil. Sellelt on näha, et hüpe 5. mai (41 päringut) ja 6. mai (6347 päringut) vahel on enam kui 150 kordne. Esiteks ilmestab see vajadust suurema maksimaalse serveri jõudluse järele, kui keskmiselt tarvis läheks. Kuid samal ajal oleks seda piiri keeruline prognoosida, sest näiteks üksikud küsitluste koostajad võivad samal ajal avalikustada küsitluse, millele vastavad kattuva lühikese ajaperioodi jooksul tuhanded inimesed. Teiseks, isegi kui serverite koormusjaotur on seadistatud automaatselt skaleeruma, võivad traditsioonilised automaatse skaleerimise süsteemid nagu AWS Auto Scaling[1] võtta kuni 5 minutit, et automaatselt uut serverit üles seada [54]. See tähendab, et kui päringute hüpe toimub lühema aja jooksul, ei pruugi rakendus uuele koormusele vastu pidada.



Joonis 5. Näide Surveeri küsitlusele vastamise hüppelisusest.

Varasema arhitektuuriga on veel ka see probleem, et kõik Surveeri alamrakendused sõltuvad samast serverist. Olgugi, et vastavalt automaatsele skaleerimisele võib neid

[1] https://aws.amazon.com/autoscaling/

servereid olla kaks või enam, mõjutavad need üksteist näiteks vea tõttu API's või suure küsitlustele vastamise koormusega nii, et igaüks neist võib korraga kõigi alamrakenduste maasoleku põhjustada. Seda probleemi saab lahendada ka ilma serverivaba arhitektuuri kasutuselevõtuta, eraldades komponendid erinevatesse serveritesse. Kuid ilma serverivaba arhitektuurita, mille puhul tuleb maksta vaid kasutatud ressursside ja aja eest, võivad muud lahendused osutuda kulukaks, sest sellisel juhul peab tasuma ka serverite eest, mida enamjaolt ei kasutata ning samuti peab arvestama serverite haldamisega seotud lisakuludega.

Tuginedes eelpool toodud faktidele leidis töö autor, et teised arhitektuurilised alternatiivid, peale serverivaba arhitektuuri koos FaaS-teenusplatvormi kasutamisega, poleks otsustamise hetkel nimetatud probleeme leevendanud.

4.3 Serverivaba arhitektuuri kasutuselevõtt

Pärast eelmises jaotises kirjeldatud puuduste sõnastamist ja serverivaba arhitektuuri omaduste tundma õppimist, sai töö autorile selgeks, et serverivaba arhitektuuri kasutuselevõtt võimaldaks kõnealustele probleemidele leevendust pakkuda. Kuna Surveeri varasem arhitektuur põhines juba erinevatel AWS pilveteenustel ja AWS Lambda on FaaS-teenusplatvormide üks tugevamaid eestvedajaid (vt jaotist 3.4.1) ning alternatiivsete pilvede omadused ei olnud vahetuseks veenvad, sai tehtud otsus jätkata AWS pilves ja võtta kasutusele selle teisi teenuseid, et saavutada soovitud serverivaba arhitektuur. Käesolevas jaotises kirjeldatakse lähemalt tekkinud valikuid ja kuidas ning mil määral uus serverivaba arhitektuur sai rakendatud.

Kuna Surveeri suurimad äririskid on seotud küsitlusele vastamise rakendusega ja rakenduse arhitektuuri kohaselt on selleks vaid mõned API-otspunktid (sisuliselt küsitluse GET-päring, vastuste saatmise POST-päring ja vastuse muutmise PUT-päring) võrreldes umbes 40 otspunktiga, mis on küsitluste loomise rakenduses, siis tundus mõistlik esialgu vaid küsitluse vastamise osa FaaS-teenusplatvormile migreerida. Selle osa migreerimist lihtsustas ka asjaolu, et küsitlustele vastamine on anonüümne ja ei vaja autentimist ega autoriseerimist. Samuti sai arvesse võetud FaaS'i külmkäivituse probleem (vt jaotis 3.3). Paari küsitlusele vastamise otspunkti on oluliselt lihtsam „soojas hoida", et külmkäivituse latentsusaega vältida ning suure tõenäosusega püsivad nad selles olekus juba orgaanilisel teel (kui küsitlustele vastatakse vähemalt 5 minutilise intervalliga).

Lisaks on Collector eesrakendus oma staatilistel failidel põhinevale olemusele lihtsal viisil serverivabasse arhitektuuri integreeritav.

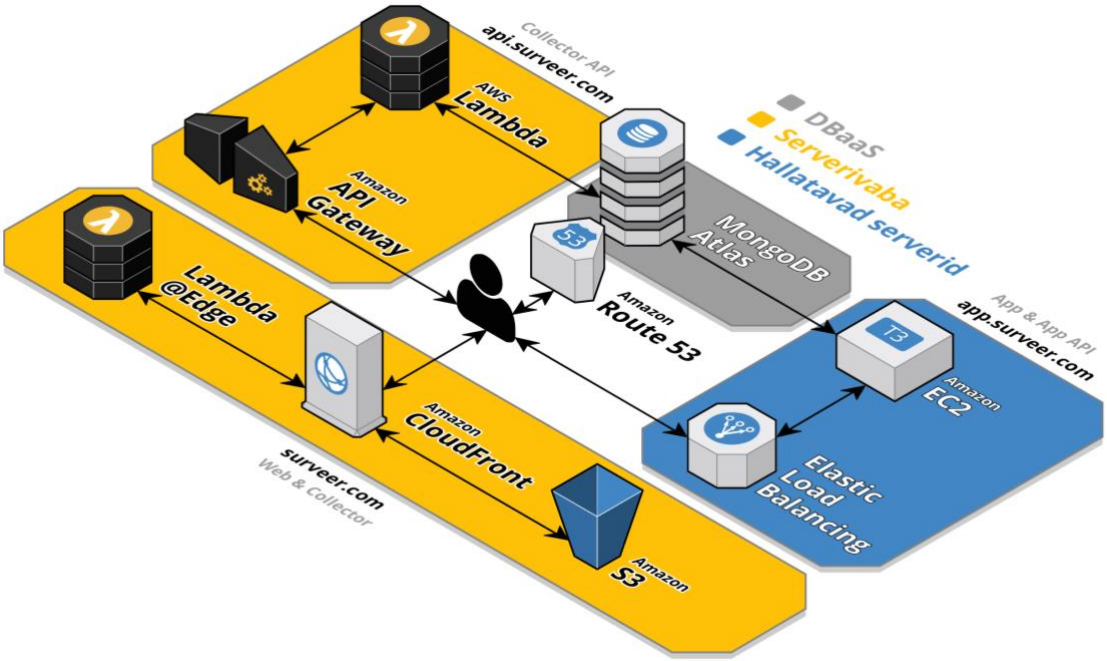
Serverivaba arhitektuuri kasutuselevõtu, juurutamise ja haldamise hõlbustamiseks otsustati kasutada Serverless raamistikku. See on kõige laialdasemalt kasutatav avatud lähtekoodiga tööriistakomplekt serverivabade rakenduste loomiseks, mis toetab kõiki peamisi pilveteenuse pakkujaid ning lisaks mitmeid avatud lähtekoodiga FaaS-platvorme [55]. Arvestades, et Surveeril on Node.js tagarakendus ja FaaS-teenusplatvormiks sai valitud AWS Lambda, siis oleks olnud peamine Serverless raamistiku alternatiiv Claudia.js[1]. Kuid Serverless raamistiku kasuks rääkis pilveteenuse pakkuja agnostiline lähenemine, st see toetab peale AWS'i ka teisi pilveteenuse pakkujaid ning kui tulevikus on soov migreerida mõnele teisele pilvele või kasutada mitut pilve korraga, siis Serverless raamistik sobib selleks endiselt. Lisaks on Serverless raamistiku arhitektuur modulaarse ülesehitusega ja omab pluginate näol laia ökosüsteemi [56], mis osutus samuti mõjusaks omaduseks.

Ülevaate Surveeri uuest arhitektuurist annab Joonis 6, kus on näha rakenduse erinevaid komponente ja andmete liikumist erinevate kasutusel olevate (peamiselt AWS) pilveteenuste vahel. Üldiselt ülesehituselt koosneb Surveeri arhitektuur nüüd viiest eraldiseisva koodibaasiga alamrakendusest, mis on jaotunud kolmele domeenile:

1. **surveer.com**
   - **Web** – avalikkusele suunatud staatiline koduleht.
   - **Collector** – AngularJS eesrakendus küsitlusele vastamiseks.
2. **api.surveer.com**
   - **Collector API** – serverivaba tagarakendus REST-otspunktidega, millega Collector eesrakendus suhtleb ja mille otspunktid on AWS Lambda funktsioonid.
3. **app.surveer.com**
   - **App** – AngularJS eesrakendus küsitluste loomiseks ja tulemuste analüüsimiseks.
   - **App API** – monoliitne Node.js tagarakendus REST-otspunktidega, millega App eesrakendus suhtleb.

[1] https://claudiajs.com/

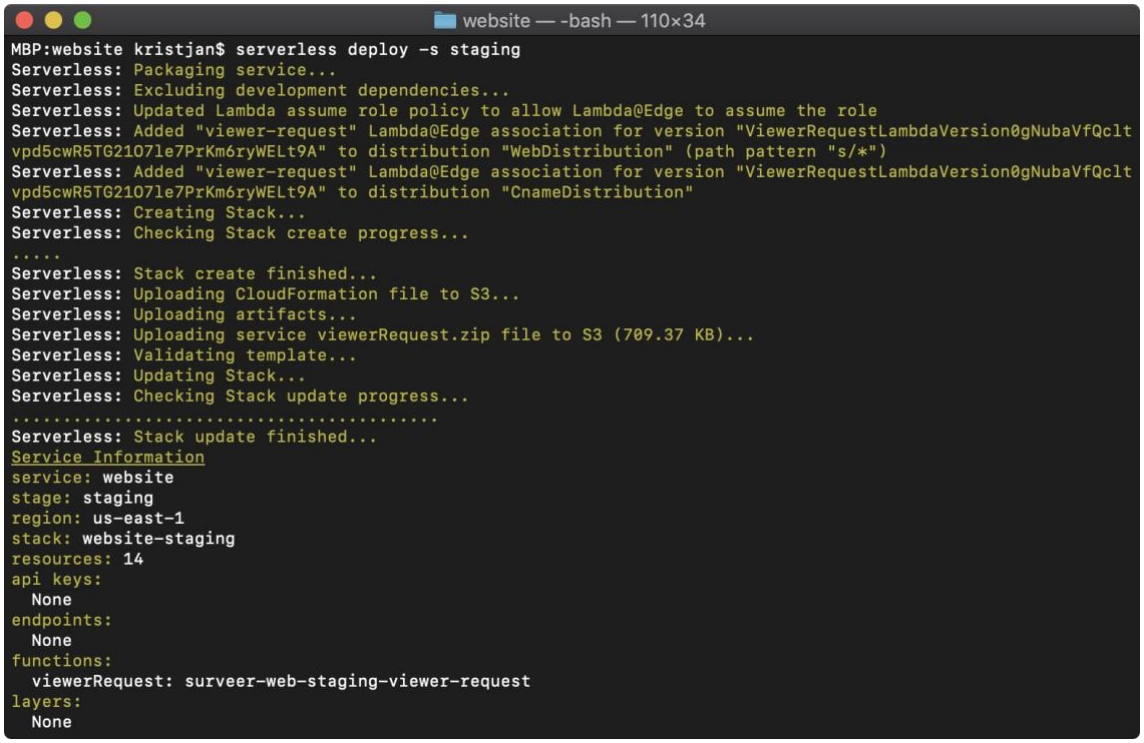
Varemalt olid kõik alamrakendused juurutatud üheskoos surveer.com juurdomeenile. Selleks, et teiste alamrakenduste jaoks saaks serverivaba arhitektuuri kasutusele võtta, pidi esimese suurema arhitektuurilise muudatusena migreerima App eesrakenduse koos App API tagarakendusega eraldi alamdomeenile app.surveer.com. Samuti lahutati Web ehk staatiline koduleht Collector rakendusest iseseisvaks komponendiks. Seda tehti selleks, et Web ja App ei mõjutaks enam teineteist ja et neid saaks nüüd eraldi juurutada. Tänu sellele on neid mõlemaid komponente lihtsam hallata ja samuti oli see eelduseks, et saaks teostada teise olulise muudatuse – migreerida juurdomeenil surveer.com kodulehe (Web) ja küsitlustele vastamise eesrakenduse (Collector) täielikult serverivabale arhitektuurile.



Joonis 6. Surveeri uus arhitektuur.

Juurdomeeni juurutamiseks sai loodud „website“ nimeline Serverless raamistiku projekt, mille konfiguratsiooni lähtekood on esitatud Lisas 2. Projekti seadistuses on defineeritud, sarnaselt AWS Elastic Beanstalk’i App projektiga, kolm keskkonda: dev, staging ja prod (vt jaotis 4.1). Serverless raamistik pakub mitmete keskkondade tuge ja tänu sellele saab projekti keskkondi juurutada ja uuendada ainsa käsurea käsuga. Näiteks Surveeri „website“ projekti staging-keskkonna puhul on selleks: „*serverless deploy -s staging*“. Joonis 7 esitab sellest ekraanipildi, mille pealt on näha ka juurutamise ajal nähtavat ülevaatlikku informatsiooni. Käsu esmasel käivitusel luuakse vastava keskkonna jaoks

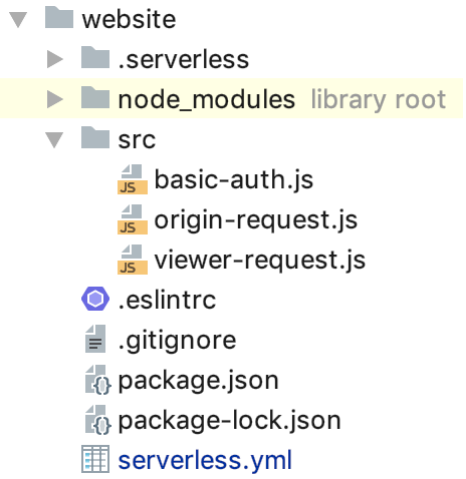
eraldi Amazon S3 failikataloogid nii Web kodulehe kui ka Collector eesrakenduse staatiliste failide hoiustamiseks ning Amazon CloudFront'i jaotis (ingl *distribution*), mis nendes paiknevaid faile vastavalt levitab. CloudFront'i jaotis on seadistatud toetama IPv6 Internetiprotokolli ning kasutama kõiki 169 [57] sisuedastusvõrgu asukohti üle maailma. Seda selle jaoks, et külastajatele saaks edastatud päritud failide koopia neile lähimast asukohast, tänu millele on latentsusaeg minimaalne. Samuti ei nõua surveer.com juurdomeeni arhitektuur pidevat süsteemiadministreerimist või serverite haldust – kõik see on AWS'i vastutusel ja tagatud 99,9% teenusetaseme lepinguga. Lisavõimaluste jaoks sai kasutatud *serverless-plugin-cloudfront-lambda-edge*[1] pluginat, tänu millele sai luua CloudFront'i Lambda@Edge funktsioone. Antud juhul said loodud kaks funktsiooni (*origin request* ja *viewer request*), et veebirobotitele (nt Google, Facebook) eelgenereeritud HTML'i serveerida. Lisaks sai loodud üks funktsioon, mis piirab dev ja staging keskkondadele ligipääsu IP-aadressi põhiselt. Serverless raamistiku juurutuskäsu järgmistel käivitustel uuendatakse ainult neid osi CloudFront'i seadistusest, mis on vahepeal muutunud ning kogemustele toetudes võib öelda, et muudatused jõustuvad üldjuhul ülemaailmselt vaid mõne minuti jooksul.



Joonis 7. Käsurea ekraanipilt "serverless deploy" käsu kasutamisest.

[1] https://github.com/silvermine/serverless-plugin-cloudfront-lambda-edge

Joonis 8 kujutab Serverless raamistiku "website" projekti failikataloogi struktuuri WebStorm[1] programmeerimiskeskkonnas. See näitab, kuivõrd lihtsa ja konkreetse ülesehitusega saab üks Serverless raamistiku projekt olla ning illustreerib selle olulisi komponente. Projekti failikataloogis paiknev alamkataloog *.serverless* on Serverless raamistiku poolt kasutusel juurutamisel tekkinud failide hoiustamiseks. Projekti üks kõige olulisem osa on konfiguratsioon (esitatud Lisas 2), mis paikneb failis *serverless.yml*. Kolme loodud Lambda@Edge funktsiooni lähtekoodid asuvad alamkataloogis *src*. Failis *.gitignore* on määratud *.serverless* ja *node_modules* kataloogid Git versioonihaldusest välja jääma. Failis *package.json* defineeritud arendustööriistad on npm käsurea rakenduse abil automaatselt laetud *node_modules* kausta ja npm tarbeks on automaatselt genereeritud *package-lock.json* fail. Lisaks varem defineeritud Serverless raamistiku pluginatele on kasutusel ka ESLint[2] arendustööriist, et teostada selle abil JavaScript failide koodistandardite kontroll (ingl *linting*), mis on seadistatud *.eslintrc* konfiguratsioonifaili alusel.

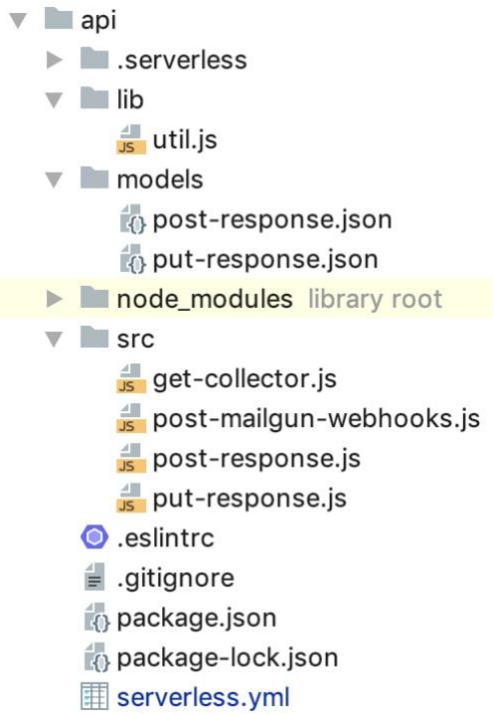


Joonis 8. Serverless raamistiku "website" projekti failikataloogi struktuur.

Collector API jaoks sai loodud „api" nimeline Serverless raamistiku projekt, mille konfiguratsiooni lähtekood on esitatud Lisas 3. Kõnealuses projektis on kasutusel kolm pluginat. Selleks, et loodud API oleks ligipääsetav soovitud domeenilt api.surveer.com, on kasutusel *serverless-domain-manager*[3] plugin. Teine erinõue oli API Gateway's

[1] https://www.jetbrains.com/webstorm/
[2] https://eslint.org
[3] https://github.com/amplify-education/serverless-domain-manager

valideerida API vastu tehtavate päringute JSON-sisu ja selleks võeti kasutusele *serverless-reqvalidator-plugin*[1], mis omakorda tingis ka kolmanda, *serverless-aws-documentation*[2], plugina kasutamise. Joonis 9 kujutab Serverless raamistiku "api" projekti failikataloogi struktuuri. Lisaks „website" projektis kirjeldatule on kõnealuses projektis ka *lib* alamkataloog *util.js* failiga, milles paiknevad funktsioonid, mis on ühised kõikidele *src* alamkataloogis paiknevatele AWS Lambda funktsioonidele (nt andmebaasiühenduse taaskasutamine). Peale selle on JSON-sisu valideerimise tarbeks loodud JSON-skeemid (ingl *JSON Schema*), mis paiknevad *models* alamkataloogis. Antud lahendus võimaldab hoida funktsioonide lähtekoodi vastava loogika võrra lühemana ning lihtsustab nende haldamist.



Joonis 9. Serverless raamistiku "api" projekti failikataloogi struktuur.

Andmete hoiustamiseks kasutatakse endiselt MongoDB Atlas andmebaasiteenust, mille klaster on juurutatud AWS Stockholmi (eu-north-1) andmekeskusesse. See sobib hästi serverivabasse arhitektuuri, sest andmebaasi serverite ülalhoid, uuendamine ja süsteemiadministreerimine on teenusepakkuja vastutusel. Teenuse tarbijana oli oluline valida pakett, mille maksimaalsete ühenduste arv ületaks soovitud maksimaalse

[1] https://github.com/RafPe/serverless-reqvalidator-plugin
[2] https://github.com/deliveryhero/serverless-aws-documentation

samaaegsete päringute arvu. Teine oluline samm oli Lambda jõudluse optimeerimine MongoDB ja Node.js koos kasutamiseks, millest on täpsemalt kirjutanud Londner [58] MongoDB ametlikus blogis. Selle peamine idee on võimaldada andmebaasiühenduste taaskasutamist Lambda konteinerites. AWS ei garanteeri Lambda konteinerite taaskasutamist, aga eksperimendid on tõestanud, et seda üldjuhul tehakse, kui konteiner leiab kasutust vähemalt iga viie minuti jooksul (vt jaotis 3.3). Kuna Surveer kasutab UptimeRobot[1] seiretarkvara, et Surveeri erinevate alamrakenduste olekul silma peal hoida, siis selle positiivse kõrvalmõjuna on igal ajahetkel vähemalt üks konteiner „soojas hoitud“.

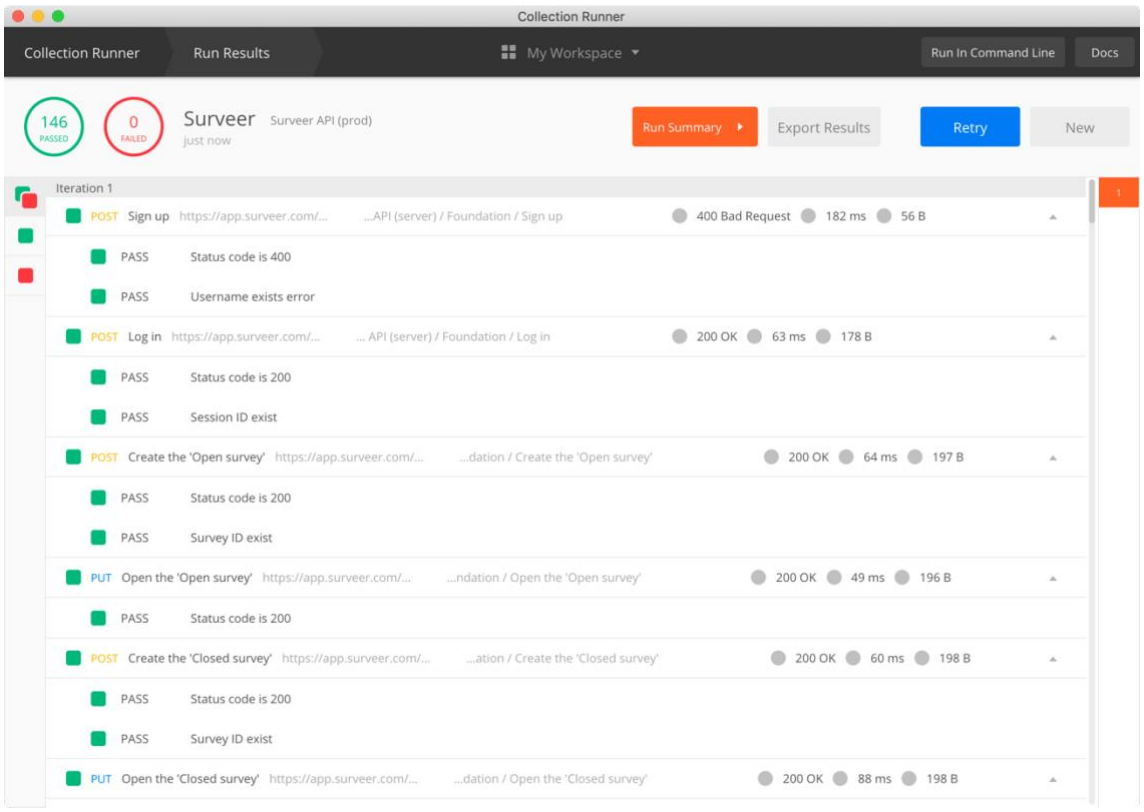
Kõikide loodud Lambda funktsioonide Node.js versiooniks sai valitud 8.10, mida AWS asus toetama aprillis 2018 [59]. Node.js 8.10 märkimisväärsed eelised, võrreldes varem toetatud versiooniga 6.10, on async/await programmeerimismustri toetamine ja kuni 20% parem jõudlus.

Selleks, et tagada migreeritud API-otspunktide nõuetele vastavus ja kvaliteet, loodi valminud Collector API jaoks komplekt Postman[2] automaatteste, mille lähtekood on esitatud Lisas 4. Iga API-otspunkti nõuded on tagatud vastavate testidega. Testid sõltuvad eeldefineeritud keskkonna põhistest muutujatest, mille hulgas on nii varjamatu sisuga muutujaid (näiteks API URL) kui ka varjatud sisuga muutujaid (näiteks test-konto ligipääsud). Iga rakenduse keskkonna (dev, staging, prod) jaoks on loodud Postman'i keskkond vastavate muutujatega. Testid algavad päringutega App API vastu, luues erinevate seadistustega küsitlusi, mida hiljem testid Collector API testimiseks kasutavad. Testide komplekt lõppeb loodud küsitluste kustutamisega. Lisaks on testides kasutusel oleval test-kontol spetsiaalne õigus rakenduse statistikat mitte mõjutada, et vältida ettevõtte võtmemõõdikute rikkumist.

Joonis 10 esitab Collector API testide läbimise tulemused läbi Postman'i graafilise kasutajaliidese. Tulemustes on näha, et kõik 146 testi läbiti edukalt ja sellest saab järeldada, et migratsioon õnnestus ning kõik API-otspunktid toimivad vastavalt ootustele. Kuid lisaks kasutajaliidesele on antud automaattestid mõeldud ka arenduskeskkonnas testimiseks ja integreerimiseks CI/CD (ingl *continuous integration, continuous delivery*)

[1] https://uptimerobot.com
[2] https://www.getpostman.com

süsteemi, kus neid on kohustatud läbima igas juurutamise etapis, et vältida rakenduse edasiarendamises regressioonivigu. Selleks kasutatakse Postman'i ametlikku „newman"[1] npm moodulit, mida saab kasutada nii käsurearakendusena kui ka liidestusena teistes Node.js rakendustes.



Joonis 10. Ekraanivaade Collector API Postman testide läbimise tulemustest.

Selleks, et juurutatud AWS Lambda funktsioonide võimalikud veateated ei jääks märkamata, võeti lisaks AWS Lambda enda kontrollpaneelile ja Amazon CloudWatch[2] teenusele kasutusele ka jaotises 3.2 kirjeldatud serverivabade tarkvarasüsteemidele seire ja tõrkeotsingu tarkvara Dashbird. Selle abil saab kohese teavituse koos täpsema informatsiooniga, kui mõne funktsiooni käivitus ebaõnnestub ja samuti annab see parema ülevaate kõikide funktsioonide käivitustest ning nende aja ja vahemälu kasutusest. Just eelnimetatud põhjustel on Dashbird osutunud töö autori kogemusel praktilisemaks tööriistaks kui CloudWatch.

[1] https://www.npmjs.com/package/newman
[2] https://aws.amazon.com/cloudwatch/

Töö autori hinnangul õnnestus Surveeri varasema monoliitse arhitektuuri migratsioon serverivabale arhitektuurile edukalt. Pärast esmast õpikõverat ja väljakutset, milleks oli FaaS paradigma ning selle omaduste ja parimate praktikate selgeks tegemine, edenes migratsioon üllatusteta. Suures plaanis tuli arhitektuurile küll komponente juurde, kuid seeläbi läks alamrakenduse hooldamine ja ülalhoid lihtsamaks. Alamrakendused Web, Collector ja Collector API on nüüd kõik läbi Serverless raamistiku lihtsal viisil juurutatavad ja hallatavad ning muudatused ühes alamrakenduses ei mõjuta enam teisi. Muudatusi ühes AWS Lambda funktsioonis saab juurutada teisi API-otspunkte mõjutamata. Loodud FaaS-funktsioonide (Collector API) kvaliteedi ja nõuetele vastavuse tagab komplekt Postman'i automaatteste. Samuti paranes oluliselt ka staatiliste failide edastamise jõudlus – eesrakenduste edastamine läbi Amazon CloudFront sisuedastusvõrgu vähendas nende kohale toimetamise latentsusaega globaalselt. Kokkuvõttes on Surveeri uus serverivaba Collector API näidanud toodangukeskkonnas enam kui poole aasta pikkusel perioodil üles soovitud töökindlust (sh küpsust, tõrketaluvust, taastuvust, töökindluse vastavust [60]) ning mis peamine – seda kõike ilma serverite pärast muretsemata.

5 Uurimistulemuste analüüs

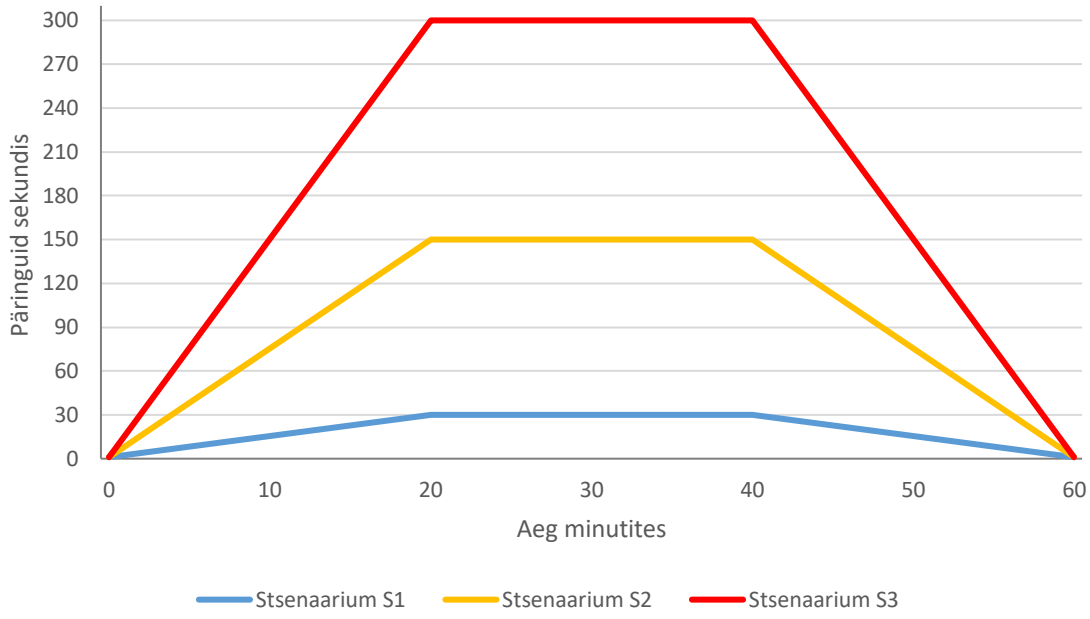
Surveeri uue serverivaba arhitektuuri valideerimiseks ja uurimisküsimustele vastamiseks viis töö autor läbi eksperimentseeriad, mis võrdlesid samadel alustel küsitlusele vastamise varasemat monoliitset ja uut serverivaba arhitektuuri. Järgnevates jaotistes kirjeldatakse kõnealuste eksperimentide metoodikat, teostatakse jõudlus- ja kuluanalüüs ning esitatakse tulemused.

Töö sissejuhatuses sõnastati kaks peamist uurimisküsimust: esimene küsimus käsitleb monoliitse ja serverivaba arhitektuuri jõudlust ning teine kulude erinevust. Selleks, et neid võrdlusi paremini samadel alustel teostada, defineeriti kolm tippkoormuse stsenaariumi, mida esitab Tabel 4.

Tabel 4. Tippkoormuse stsenaariumid arhitektuuride võrdluseks.

Stsenaariumi ID	**Tippkoormuse API-päringuid sekundis**	**Keskmiselt API-päringuid sekundis tunni jooksul**	**API-päringuid tunnis**
S1	30	20	72 000
S2	150	100	360 000
S3	300	200	720 000

Kõnealused stsenaariumid on kavandatud kui võimalikud küsitlustele vastamise tippkoormused ühe tunni jooksul. Kõigil juhtudel on planeeritud koormuse kasv esimese 20 minuti jooksul ühest päringust sekundis kuni stsenaariumis seatud tippkoormuseni (näiteks S3 puhul 300 päringuni sekundis), mis kestab 20 minutit ning seejärel langeb sujuvalt viimase 20 minuti jooksul tagasi ühe päringuni sekundis. Seepärast on keskmine koormus sekundis kolmandiku võrra väiksem kui tippkoormus sekundis. Stsenaariumide koormust ajas illustreerib joondiagrammiga Joonis 11. Lisaks on arvesse võetud, et keskmine Surveeri küsitluse lõpetamise protsent on umbes 50, st kaks korda rohkem avatakse küsitluse ankeeti (ja tehakse GET-päringuid) kui sellele lõpuks vastatakse.



Joonis 11. Stsenaariumide koormuste võrdlus.

Aastal 2018 oli küsitlustarkvara turuliidri SurveyMonkey avaldatud andmete [61] põhjal nende platvormis küsitlustele vastajate arv päevas üle kahe miljoni, mis lineaarse koormuse puhul oleks minimaalselt umbes 23 vastajat sekundis. Kuigi kõnealuste eksperimentide eesmärk on võimalike tippkoormuste (mitte pideva lineaarse koormuse) testimine ja analüüsimine, annab see informatsioon siiski teatava perspektiivi ühe eduka globaalse küsitlustarkvara oodatavast koormusest.

Võrdluse all olev varasem küsitlustele vastamise arhitektuur baseerus AWS Elastic Beanstalk'i abil juurutatud c5.large tüüpi Amazon EC2 pilveserveritel ja uus arhitektuur Serverless raamistikuga juurutatud AWS Lambda platvormil, kasutades Amazon API Gateway'd. EC2 Node.js versioon oli 8.15.1 ja Lambda'l 8.10. Seejuures olid AWS Lambda funktsioonid seadistatud kasutama 128 MB mälumahtu. Mõlema arhitektuuri puhul kasutati MongoDB Atlas andmebaasiteenust, mis oli juurutatud samasse AWS Stockholm (eu-north-1) andmekeskusesse, sarnaselt teiste arhitektuuri komponentidega. Varasemast arhitektuurist annab täpsema ülevaate jaotis 4.1 ja uuest 4.3.

5.1 Jõudlusanalüüs

Kavandatud jõudlustestimise skoobiks on küsitlusele vastamise API-otspunktide testimine (JSON sisuga GET ja POST päring). Eksperimentide eesmärk on mõõta päringute õnnestumist ja reaktsiooniaega (ingl *response time*) tippkoormusega perioodide ajal. Reaktsiooniaja mõõtmine on oluline, sest see illustreerib serveri võimet koormusega toime tulla ja samuti ka sisulisel põhjusel, kuna see on peamine tegur, mis põhjustab külastajate kaotust. Seda ilmestab BBC uue veebilehe juhtumiuuring [62], mille käigus leiti, et nad kaotasid 10% kasutajatest iga sekundiga, mis kulus nende saidi laadimisele. Kavandatud jõudlustestid on kõik planeeritud kestma 60 minutit, mille vältel testitakse kolme erinevat tippkoormust.

Kui soov oleks simuleerida täielikku küsitlusele vastamise protsessi, siis nendele kahele API-päringule peaks eelnema eesrakenduse staatiliste HTML, CSS ja JavaScript failide allalaadimine. Kuid võttes arvesse, et varasema arhitektuuri puhul jagati neid staatilisi faile samast serverist, mille ressursse kasutas ka küsitlusele vastamise API, mõjutaks see olulisel määral varasema arhitektuuri jõudlustestimise tulemusi. Seevastu toimub uuel arhitektuuril staatiliste failide jagamine läbi Amazon S3'e ja Amazon CloudFront sisuedastusvõrgu, mille vaikimisi limiit ühe jaotise kohta on sada tuhat päringut sekundis [63], mistõttu oleks antud lahenduse puhul staatiliste failide jõudlustestimine samuti ebavajalik.

Jõudlustestimise läbiviimiseks valiti rakendus Apache JMeter [64] 5.1.1, mis on laialdaselt kasutusel olev avatud lähtekoodiga tarkvara veebirakenduste koormustestide tegemiseks ja jõudluse mõõtmiseks. Teiste seas kasutavad JMeter'it ka AOL, Orbitz ja Lufthansa [65]. Kuna testimise eesmärk oli mõõtmised samadel alustel läbi viia nii uue kui ka vana arhitektuuri peal, siis platvormide sisest või logidel põhinevat statistikat poleks saanud üks-ühele võrrelda ning seepärast osutus JMeter sobivaks tööriistaks. Testide läbiviimiseks juurutati JMeter c5.large tüüpi Amazon EC2 pilveserverisse, mis paiknes testitavate osapooltega samas AWS Stockholm (eu-north-1) andmekeskuses.

Autori poolt loodud JMeter'i testplaani (ingl *test plan*) lähtekood on esitatud Lisas 5. Kõnealune testplaan simuleerib küsitlusele vastamisel toimuvaid REST API HTTP-päringuid (üle HTTPS-protokolli). Testimise otstarbeks loodi keskmise pikkusega (10 küsimust) küsitlus. Vana ja uue arhitektuuri testid erinesid teineteisest ainult

keskkonnamuutujate poolest, mida sai loodud testi käivitamisel määrata käsurealt. Testseeriate lihtsamaks läbiviimiseks lõi töö autor *shell*-skripti, mille kasutamisel tuli määrata testitav arhitektuur ja tippkoormuse päringud sekundis, misjärel käivitas skript JMeter'i testplaani. Skripti lähtekood ja näide selle kasutamisest on esitatud Lisas 6. Tänu sellele sai testimised lihtsal viisil cron'iga ajastada. JMeter'i testplaani kood oli mõlema arhitektuuri jaoks sama. Test algas küsitluse JSON-sisu alla laadimisega, mis on küsitlusele vastamise API GET-päring. Tavapärasele kasutaja käitumisele järgneks küsitluse vastamisele kuluv aeg, misjärel vastaja saadaks oma vastused (POST-päring JSON-sisuga, milles on vastused). Kuid võttes arvesse, et antud jõudlustestimise eesmärk oli mõõta ning võrrelda arhitektuuride päringute õnnestumist ja reaktsiooniaega kindlal tippkoormusel, ei olnud praktiline imiteerida küsitlusele vastamiseks kuluvat aega, sest keskmine küsitluse vastamisele kuluv aeg Surveeris on 223 sekundit. Küsitlusele vastamise POST-päring tehti lihtsuse huvides kohe pärast GET-päringut, ilma ajalist viidet lisamata. Maksimaalseks reaktsiooniajaks seati 3000 ms – st päringud, mis kestsid kauem, loeti ebaõnnestunuks.

Töö autor viis läbi 18 tunniajast jõudlustestimist – kolm seeriat iga stsenaariumi ja mõlema arhitektuuri kohta. Testid ajastati nii, et iga Lambda + API Gateway testimise vahel oleks vähemalt üks koormuseta tund, et iga testimine algaks nn külmkäivituse olukorras. Viited testimistulemuste toorandmetele (mida on kokku ligi 1 GB) on esitatud Lisas 7. Testimistulemuste andmetöötluse jaoks kasutati JMeter'i pluginaid ja graafilist kasutajaliidest. Tulemuste lugemises peab arvesse võtma, et JMeter ei pruugi hoida igal ajahetkel seatud koormustingimusi 100% täpsusega. Seepärast on üldjuhul tulemuste päringuid sekundis statistikas sees väike erinevus võrreldes eeldatavaga ning sellest tingituna on ka täpne päringute koguarv varieeruv.

Stsenaariumi S1 tippkoormuseks oli määratud 30 API-päringut sekundis. Mõlema arhitektuuri puhul tehti testimiste käigus kokku umbes 214 tuhat päringut (kolm seeriat), millest oodatavalt umbes kaks kolmandikku olid küsitluse GET-päringud ja üks kolmandik POST-päringud. Tunniajaste testseeriate keskmine läbilaskevõime oli mõlema arhitektuuri puhul ootuspäraselt kokku umbes 20 päringut sekundis. EC2'l põhinev arhitektuur pidas koormusele vastu ühe serveriga, automaatne skaleerimine testimise käigus ühtegi serveri instantsi koormuse tasakaalustajasse (ALB) juurde ei lisanud. Kummalgi arhitektuuril antud koormusega probleeme polnud, mõlema puhul oli veaprotsent 0,00. Võttes arvesse FaaS'i külmkäivituse omapära, oli maksimaalne

reaktsiooniaeg Lambda'l ootuspäraselt oluliselt kõrgem (2489 vs 374 ms). Samuti oli Lambda'l kõrgem reaktsiooniaegade standardhälve (26,18 vs 18,61 ms), kuid autori üllatuseks ilmnes jõudlustestimise tulemustest, et Lambda'l põhinev arhitektuur näitas võrreldes EC2'ga kokkuvõttes madalamat keskmist reaktsiooniaega (34 vs 39 ms). Kokkuvõtlikud tulemused esitab Tabel 5 ja tulemuste graafikud on välja toodud Lisas 8.

Tabel 5. Stsenaariumi S1 jõudlustestimise tulemused.

	EC2 (1 instants) + ALB			**Lambda + API Gateway**		
	GET	**POST**	**Kokku**	**GET**	**POST**	**Kokku**
Päringute arv	143 567	70 975	214 542	143 181	70 944	214 125
Keskmine reaktsiooniaeg (ms)	32	54	**39**	27	47	**34**
Miinimum (ms)	16	28	16	14	18	14
Maksimum (ms)	374	177	374	2489	1938	2489
Standardhälve (ms)	13,79	18,61	18,61	21,38	29,59	26,18
Veaprotsent	0,00	0,00	**0,00**	0,00	0,00	**0,00**
Läbilaskevõime (päringut sekundis)	13,29	6,60	**19,85**	13,25	6,59	**19,81**
Vastuvõetud KB/s	19,51	4,34	23,84	39,26	1,25	40,51
Saadetud KB/s	2,50	3,49	5,99	2,55	3,49	6,03

Stsenaariumile S2 oli määratud võrreldes S1'ga viis korda suurem tippkoormus, 150 API-päringut sekundis. Mõlema arhitektuuri puhul tehti testimiste käigus kokku umbes 1 miljon 80 tuhat päringut, millest oodatavalt umbes kaks kolmandikku olid küsitluse GET-päringud ja üks kolmandik POST-päringud. Tunniajaste testseeriate keskmine läbilaskevõime oli mõlema arhitektuuri puhul ootuspäraselt kokku umbes 100 päringut sekundis. EC2'l põhinev arhitektuur lisas automaatse skaleerimisega koormuse tasakaalustajasse (ALB) juurde 3 serveri instantsi, st tippkoormuse ajal oli kasutusel kokku 4 instantsi. Maksimaalsetest reaktsiooniaegadest on märgata, et mõlemad EC2 otspunktid ja Lambda GET-otspunkt olid üle 3000 ms, mis viitab sellele, et mõnel juhul oli tekkinud probleeme päringutele vastamisega. Kuid veaprotsent oli mõlemal juhul 0,00, seega märkimisväärseid probleeme antud koormusega ei ilmnenud. Erinevalt stsenaariumi S1 tulemustest, oli antud juhul EC2'l suurem standardhälve (27,61 vs 24,89), sellest võib järeldada, et Lambda pidas antud koormusel stabiilsemalt vastu.

Kõnealune stsenaarium näitas mõlema arhitektuuri vahel võrdset keskmist reaktsiooniaega (35 ms). Kokkuvõtlikud tulemused esitab Tabel 6 ja tulemuste graafikud on välja toodud Lisas 9.

Tabel 6. Stsenaariumi S2 jõudlustestimise tulemused.

	EC2 (4 instantsi) + ALB			**Lambda + API Gateway**		
	GET	**POST**	**Kokku**	**GET**	**POST**	**Kokku**
Päringute arv	721 107	359 122	1 080 229	720 956	359 159	1 080 115
Keskmine reaktsiooniaeg (ms)	29	48	**35**	28	49	**35**
Miinimum (ms)	14	21	14	14	17	14
Maksimum (ms)	3136	3137	3137	3010	2281	3010
Standardhälve (ms)	22,72	31,77	27,61	20,46	26,81	24,89
Veaprotsent	0,00	0,00	**0,00**	0,00	0,00	**0,00**
Läbilaskevõime (päringut sekundis)	66,79	33,38	**99,99**	66,72	33,40	**99,96**
Vastuvõetud KB/s	97,99	21,95	119,92	197,72	6,34	204,04
Saadetud KB/s	12,56	17,70	30,23	12,83	17,65	30,46

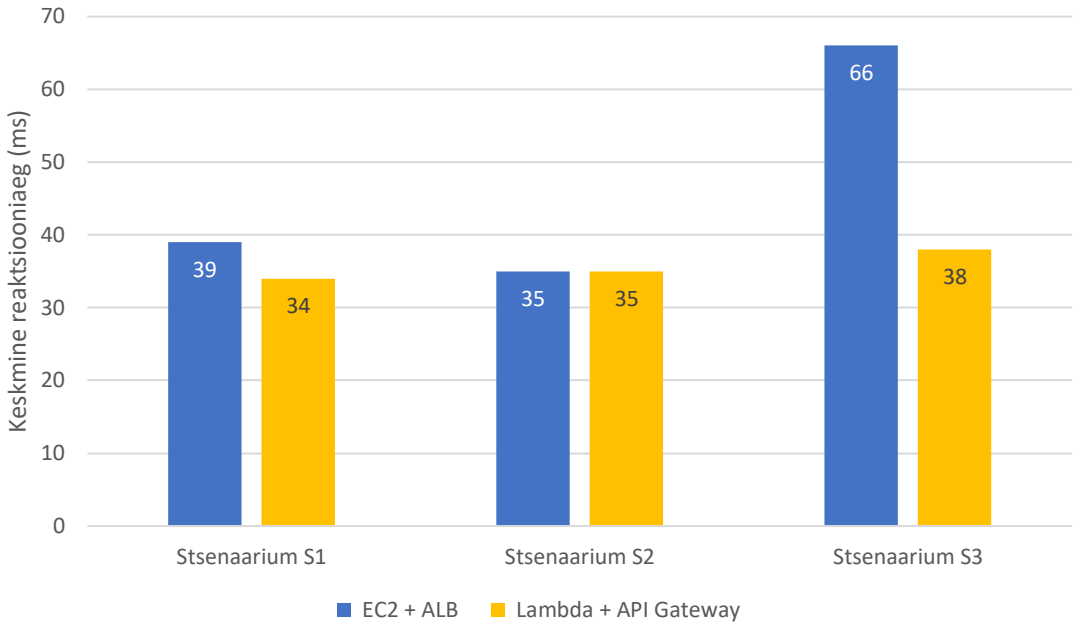
Stsenaariumile S3 oli määratud võrreldes S1'ga kümme korda ja võrreldes S2'ga kaks korda suurem tippkoormus, 300 API-päringut sekundis. EC2'l põhinev arhitektuur lisas automaatse skaleerimisega koormuse tasakaalustajasse (ALB) juurde 7 serveri instantsi, st tippkoormuse ajal oli kasutusel kokku 8 instantsi. EC2 puhul tehti testimiste käigus kokku umbes 2 miljonit 159 tuhat ja Lambda puhul umbes 2 miljonit 155 tuhat päringut. Sarnaselt eelnevatele stsenaariumidele jaotusid päringud mõlema arhitektuuri puhul oodatavalt nii, et umbes kaks kolmandikku olid küsitluse GET-päringud ja üks kolmandik POST-päringud. Tunniajaste testseeriate keskmine läbilaskevõime oli mõlema arhitektuuri puhul ootuspäraselt kokku umbes 200 päringut sekundis. Antud stsenaariumis ilmnes EC2 arhitektuuriga probleeme. Olgugi, et automaatne skaleerimine oli seadistatud võimalikult kiirelt ja võimalikult lühikese kontrollaja tagant (1 minut), tekkis enne teise serveri automaatset juurutamist päringutele vastamisega probleeme. Graafikutelt (vt Lisa 10) on hästi näha probleemset perioodi umbes kuuenda minuti juures kõigil kolmel EC2 testil. Lambda puhul taolisi skaleerimise probleeme polnud. Kui EC2 veaprotsent kokku oli 0,95, siis Lambda'l oli selleks endiselt 0,00. EC2 standardhälve oli

seekord 1000% suurem kui Lambda'l (282,78 vs 25,7 ms). See näitab kahte asja – esiteks EC2 arhitektuuri probleeme antud koormusel ja teiseks seda, et Lambda standardhälve (ja koormustaluvus) on püsinud üsna stabiilselt iga stsenaariumi korral. Kõnealuse stsenaariumiga ilmnes arhitektuuride keskmiste reaktsiooniaegade vahel suurem erinevus – Lambda näitas 42% madalamat aega (66 vs 38 ms). Lõppkasutaja nende reaktsiooniaegade vahet tõenäoliselt ei tunnetaks, kuid see tulemus peegeldab eelnimetatud fakti, et EC2 ei suutnud kõigile päringutele edukalt vastata. Kokkuvõtlikud tulemused esitab Tabel 7 ja tulemuste graafikud on välja toodud Lisas 10.

Tabel 7. Stsenaariumi S3 jõudlustestimise tulemused.

	EC2 (8 instantsi) + ALB			**Lambda + API Gateway**		
	GET	**POST**	**Kokku**	**GET**	**POST**	**Kokku**
Päringute arv	144 1241	717 552	2 158 793	1 437 946	716 949	2 154 895
Keskmine reaktsiooniaeg (ms)	59	79	**66**	31	52	**38**
Miinimum (ms)	4	4	4	14	17	14
Maksimum (ms)	3116	3105	3116	3011	2468	3011
Standardhälve (ms)	281,43	284,99	**282,78**	22,15	26,20	**25,70**
Veaprotsent	0,92	1,02	**0,95**	0,00	0,00	**0,00**
Läbilaskevõime (päringut sekundis)	133,34	66,66	**199,73**	133,08	66,60	**199,43**
Vastuvõetud KB/s	326,39	74,05	400,3	219,08	7,02	226,10
Saadetud KB/s	41,49	58,43	99,81	14,21	19,56	33,77

Ülevaate arhitektuuride tippkoormuste jõudlustestimise keskmistest reaktsiooniaegadest esitab Joonis 12. Varasem arhitektuur (Amazon EC2 + ALB) näitas võrreldes uue arhitektuuriga (AWS Lambda + API Gateway) kahe esimese stsenaariumiga (S1 ja S2) sarnaseid reaktsiooniaegu. Huvitav anomaalia on EC2 arhitektuuri puhul see, et S2'e keskmine reaktsiooniaeg on võrreldes S1'ga madalam (35 vs 39 ms). Nende vahe on küll marginaalne, aga tõenäoline põhjus selleks on lihtsalt sel periood kasutusel olnud konkreetse riistvara võimekus. Stsenaariumi S3 (tippkoormusega 300 päringut sekundis) puhul näitas uus arhitektuur endist stabiilsust, samal ajal kui varasema arhitektuuriga hakkas ilmnema skaleeruvusega probleeme. S3 puhul näitas uus arhitektuur võrreldes varasemaga 42% madalamat keskmist reaktsiooniaega (66 vs 38 ms).



Joonis 12. Tippkoormuste stsenaariumide keskmise reaktsiooniaja võrdlus.

Jõudlustestimise tulemuste abil sai töö autori poolt püsitatud esimene, jõudlust võrdlev uurimisküsimus, vastuse – mõlemad arhitektuurid näitasid tippkoormustes üldjoontes võrdväärset päringute reaktsiooniaega, kuid erinevus seisnes suuremal koormusel AWS Lambda + Amazon API Gateway kiiremas ja stabiilsemas skaleeruvuses võrreldes Amazon EC2 + Application Load Balancer arhitektuuriga. Sellest järeldub, et serverivaba arhitektuur (Lambda + API Gateway näitel) on traditsioonilise arhitektuuri (EC2 + ALB näitel) kõrval reaktsiooniaega arvestades igati võrdväärne alternatiiv. Võrreldes traditsioonilise arhitektuuriga näib serverivaba arhitektuur olevat sobivam rakendustele, mis on pigem ettearvamatu koormusega ja mille tippkoormuse kasv võib toimuda minutite jooksul.

5.2 Kuluanalüüs

Serverivaba arhitektuuri üks peamisi omadusi on kasutuspõhine hinnastamine. Seda kontseptsiooni on lihtne mõista, kuid mitte nii lihtne alternatiividega võrrelda. Käesoleva töö teine uurimisküsimus oli leida serverivaba arhitektuurile migreeritud API pilvekulude erinevused võrreldes monoliitse arhitektuuriga. Antud jaotis esitab sellele küsimusele vastuse kuluanalüüsi tulemuste põhjal. Esimene eesmärk on leida ja võrrelda mõlema arhitektuuri kulu päringu kohta. Teine eesmärk on leida, milline arhitektuur on kulutõhusam vastavalt koormuse kasvule, tuues võrdlusesse sisse ka kolmanda AWS

Lambda + ALB (Application Load Balancer) tüüpi arhitektuuri. Arvutustes kasutatud hinnad on võetud AWS Stockholmi (eu-north-1) andmekeskuse hinnakirjast (seisuga 22.04.2019) ning AWS'i igakuiseid soodustusi (ingl *free tier*) pole parema võrreldavuse huvides kalkulatsioonidesse sisse arvestatud.

Varasema arhitektuuri puhul kasutusel olnud nõudlusel (ingl *on-demand*) c5.large tüüpi EC2 pilveserveri instants maksab 0,091 USD tunnis [66]. Kuid lisaks instantsi põhikulule tuleb tasuda veel väljaminevate andmete (sissetulevad andmed on hetkel tasuta) kettamahu (Amazon Elastic Block Store, EBS) ja koormuse tasakaalustaja (Elastic Load Balancing, ELB) eest. Vastavalt kasutusele võib tekkida kulu ka Amazon CloudWatch monitoorimise eest, kuid antud kuluanalüüs seda ei sisalda. EC2 väljaminevate andmete hind Internetti kuni 10 TB'ni maksab 0,09 USD GB-kohta, kulu arvutamiseks saab kasutada jõudlustestimisest pärinevat informatsiooni, kui ühe päringu kohta saadeti keskmiselt välja 1,2 KB andmeid. Kasutusel oleva gp2 tüüpi EBS SSD hind on 0,1045 USD GB-kuu kohta [67]. Igale EC2'e instantsile on eraldatud 10 GB kettamahtu. ELB on seadistatud kasutama ALB tüüpi koormuse tasakaalustajat, põhikuluga tunnis 0,02394 USD, millele arvestatakse lisaks tarbimispõhine LCU lisatasu. LCU hind on 0,0076 USD tunnis ühe ühiku kohta. Käesoleva kuluanalüüsi lihtsustamiseks on eeldatud, et LCU hinnaarvutuse arvestatav komponent on uued ühendused (1 LCU = 25 uut ühendust) ning iga API-päring on uus ühendus.

Surveeri uue arhitektuuri võtmekomponendi AWS Lambda puhul tuleb maksta ainult selle eest, mida kasutatakse. Raha võetakse päringute arvu ja aja eest, mis koodi täitmisele kulub [68]. Ajaline kestus arvutatakse alates koodi täitmise algusest kuni vastuse või vea esitamiseni, ümardades aega ülespoole lähima 100 millisekundini. Hind sõltub funktsioonile eraldatud mälu suurusest. Lambda päringu hind on 0,0000002 USD, millele lisandub 0,00001667 USD iga GB-sekundi kohta [69]. Kõnealuses arhitektuuris kasutusel oleva 128 MB mäluga funktsiooni jaoks on selleks 0,0000002083750 USD iga 100 ms kohta. Kuna jõudlustestimise keskmine reaktsiooniaeg oli iga stsenaariumi puhul alla 100 ms, arvestatakse kuluanalüüsis iga päringu jaoks 100 ms kulu. Amazon API Gateway hind esimese 333 miljoni API-päringu jaoks on 0,0000035 USD päringu kohta, millele lisandub EC2 hinnakirja alusel andmeedastuse tasu. API Gateway API-päringute puhverdamisteenus antud arhitektuuri puhul kasutusel pole.

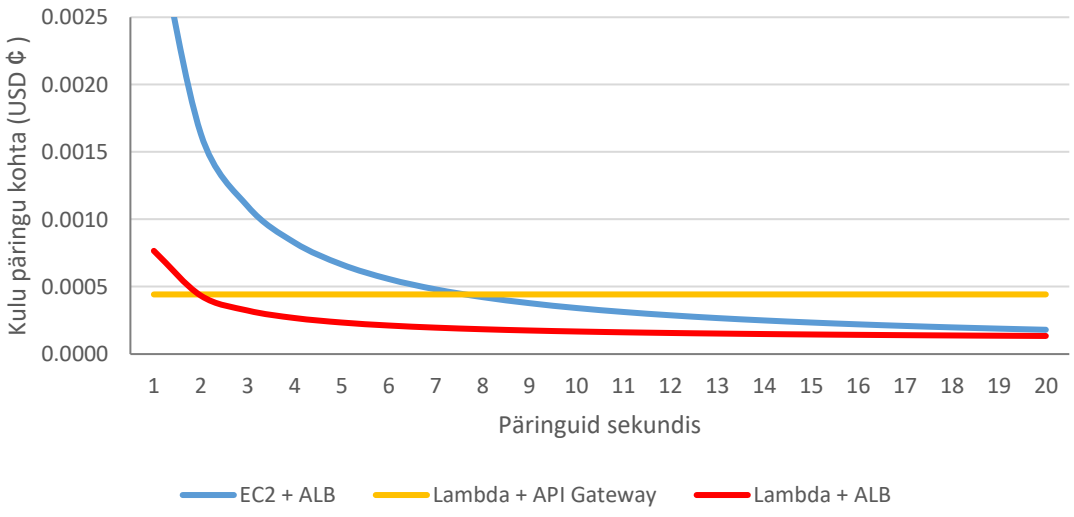
Arhitektuuride kulude võrdluse tippkoormuste stsenaariumide põhiselt esitab Tabel 8. Selles on välja toodud iga stsenaariumi kohta (mis ka jõudlustestimise aluseks oli) illustreerivalt keskmine päringute arv sekundis ja tunnis, EC2 c5.large instantside arv (empiiriliselt jõudlustestimisest) ning kalkuleeritud kulu tunnis ja päringu kohta mõlema arhitektuuri puhul (USD sentides). Varasema traditsioonilise arhitektuuri puhul (EC2 + ALB) on näha, et vastavalt koormuse kasvule läheb kulu päringu kohta väiksemaks. Kuna uue serverivaba arhitektuuri (Lambda + API Gateway) kulu on kasutuspõhine ja fikseeritud päringu kohta, siis see on iga stsenaariumi puhul sama (0 päringu puhul oleks 0). Stsenaariumi S1 puhul on EC2 + ALB umbes 2,4 korda soodsam kui Lambda + API Gateway ning stsenaariumi S3 puhul juba umbes 3,7 korda soodsam.

Tabel 8. Arhitektuuride kulude võrdlus tippkoormuste stsenaariumide põhiselt.

	Stsenaarium S1	**Stsenaarium S2**	**Stsenaarium S3**
Päringute arv sekundis	20	100	200
Päringute arv tunnis	72 000	360 000	720 000
EC2 instantse (c5.large)	1	4	8
EC2 + ALB kulu tunnis (USD)	0,12987	0,46114	0,89835
Lambda + API Gateway kulu tunnis (USD)	0,31882	1,59412	3,18825
EC2 + ALB kulu päringu kohta (USD¢)	**0,00018**	**0,00013**	**0,00012**
Lambda + API Gateway kulu päringu kohta (USD¢)	0,00044	0,00044	0,00044

Surveeri küsitlusele vastamise API uue arhitektuuri loomisel kasutati AWS Lambda't koos Amazon API Gateway'ga. Kõnealune arhitektuur on üles näidanud head töökindlust. Jõudlusanalüüs tõestas, et võrreldes EC2 arhitektuuriga kannatas see tippkoormust stabiilselt ja probleemivabalt. Kuid eelnevast kulude võrdlusest (vt Tabel 8) selgus, et tippkoormustel on uus arhitektuur varasemast kordades kallim. 2018. aasta lõpus avalikustas AWS esmakordselt võimaluse kasutada Lamba't API Gateway asemel koos Application Load Balancer'iga (ALB), mis muuhulgas lõi soodsama alternatiivi serverivaba API loomiseks. Sel põhjusel on huvitav uurida, milline on kõnealuse arhitektuuri kulu võrreldes kahe teise töös vaadeldud arhitektuuriga.

Joonis 13 esitab joondiagrammi abil arhitektuuride kulude võrdluse päringu kohta, keskendudes koormustele kuni 20 päringut sekundis. Võrdluse all on kolm arhitektuuri: traditsiooniline EC2 + ALB, serverivaba Lambda + API Gateway ning alternatiivne Lambda + ALB. Tulemustest on näha, et Lambda + API Gateway arhitektuur on kulutõhusam, kui koormus on madalam kui 7,7 päringut sekundis. Sellest hetkest on EC2 + ALB kulu päringu kohta väiksem, kui Lambda + API Gateway muutumatu kulu 0,000000443 USD. Täpselt sama tulemuseni jõudsid ka Crane ja Lin [7] oma juhtumiuuringus (vt peatükk 2). Töö autorit üllatav avastus seisneb hoopis Lambda + API Gateway ja Lambda + ALB arhitektuuride võrdluses – tulemustest selgub, et kui koormus on suurem kui 2 päringut sekundis, muutub Lambda + ALB kulutõhusamaks. Kalkulatsioonides kasutati ALB'i kulu hindamiseks mõlemal juhul sama loogikat. Koormusel 20 päringut sekundis on Lambda + API Gateway kulu endiselt 0,000000443 USD, EC2 + ALB kulu on 0,000000180 USD ning Lambda + ALB kulu on 0,000000135 USD. See tähendab, et sellisel koormusel on viimane esimesest üle 3 korra soodsam. Kuid siinkohal tasub ära märkida, et võrreldes API Gateway kasutamisega, mille eest ei pea midagi maksma kui päringud puuduvad, peab ALB'i kasutamise eest tasuma põhitasu ka juhul, kui reaalne tarbimine puudub (0,02394 USD tunnis). Seetõttu, toetudes kuluanalüüsi tulemustele, on 0-2 päringut sekundis kõige kulutõhusam Lambda + API Gateway arhitektuur ja alates kahest päringust sekundis Lambda + ALB arhitektuur. Tulemuste tõlgendamisel peaks ka meeles pidama, et EC2 puhul arvutati kulu nõudlusel (ingl *on-demand*) pilveserverite hinnakirja alusel. See võib olla aga võrreldes EC2 reserveeritud (ingl *reserved*) pilveserveriga 40-60% kulukam [70].



Joonis 13. Arhitektuuride kulude võrdlus.

Töö autori poolt püstitatud pilvekulusid võrdlev uurimisküsimus leidis kuluanalüüsi tulemuste abil mitu vastust. Kahe arhitektuuri võrdluses oli EC2 + ALB arhitektuur iga tippkoormuse stsenaariumi puhul kulutõhusam. Kuid lisades üldisesse võrdlusesse ka kolmanda, Lambda + ALB arhitektuuri, selgus, et koormusel 0 kuni 2 päringut sekundis on kõige kulutõhusam Lambda + API Gateway arhitektuur ning alates kahest päringust sekundis on selleks Lambda + ALB arhitektuur. Samuti tasub arvestada asjaolu, et serverivaba arhitektuuri kasutuselevõtt võib kõrvaldada vajaduse teatud IT-rollide (nt süsteemiadministraatori) järele, misläbi võib ettevõtte saavutada märkimisväärse kokkuhoiu tööjõukulude osas (vt jaotis 3.2).

Põhinedes kuluanalüüsi tulemustele ja võttes arvesse ka jõudlusanalüüsi tulemusi, jõudis töö autor järeldusele, et teostatud Surveeri migratsioon serverivabale arhitektuurile oli äriliselt põhjendatud. Edasiarenduste osas selgus ka see, et kui Surveeri keskmine küsitlustele vastamise koormus ületab kahte päringut sekundis, siis on pilveteenuste kulu silmas pidades mõistlik migreerida kasutusel olevad API-otspunktide Lambda funktsioonid API Gateway asemel Application Load Balancer'i kasutama. Töö autorile on uute projektide puhul serverivaba arhitektuur eelistatud alternatiiv ning soovitab seda ka teistele (eriti alustavatele) ettevõtetele.

6 Kokkuvõte

Käesoleva magistritöö põhieesmärgiks oli uurida ja analüüsida mikroteenustel põhineva serverivaba arhitektuuri kasutamise eeliseid ja puudusi võrreldes monoliitse arhitektuuriga. Töö autor tegi teemavaldkonna ülevaate seonduvatest uurimustest, mis näitasid suurt huvi kasvu serverivaba valdkonna vastu teadustöödes ning viitasid erinevates uurimistöödes saadud positiivsetele tulemustele serverivaba arhitektuuri osas.

Serverivaba arhitektuuri (keskendudes FaaS'ile) eelisteks leiti suurem produktiivsus, automaatne skaleerimine, kulude kokkuhoid ja „rohelisem" andmetöötlus. Peamisteks puudusteks osutusid tootjalukustus, turvalisusküsimused, funktsioonide oleku puudumine, külmkäivitus ning piiratud käitusaeg ja mälumaht. Põhieesmärki toetava eesmärgina kirjeldati küsitlustarkvara Surveer varasemat arhitektuuri ja dokumenteeriti serverivaba arhitektuuri kasutuselevõttu, migreerides Surveeri kõrgema äririskiga monoliitse REST API otspunktid serverivabadeks mikroteenusteks, kasutades selleks Serverless raamistikku koos AWS Lambda platvormiga.

Töö eesmärgi saavutamiseks püstitati uurimisküsimused serverivabale arhitektuurile migreeritud API jõudluse ja pilvekulude võrdluse kohta. Nendele aitasid vastuseid leida jõudlustestimise seeriad, millega võrreldi Surveeri varasemat ja uut arhitektuuri ühise testide komplekti alusel. Jõudlusanalüüsi tulemustes näitasid mõlemad arhitektuurid tippkoormustes võrdväärset reaktsiooniaega, kuid erinevus seisnes tippkoormusel AWS Lambda stabiilsemas skaleeruvuses. Kuluanalüüsi tulemuste põhjal selgus, et kahe arhitektuuri võrdluses oli traditsiooniline serveritega arhitektuur kulutõhusam alates koormusest 7,7 päringut sekundis. Kuid kõrvutades eelnevatega kolmandat serverivaba arhitektuuri, milles Lambda platvormi kasutatakse API Gateway asemel Application Load Balancer'iga (ALB), ilmnes, et koormusel 0 kuni 2 päringut sekundis oli kulutõhusam Lambda koos API Gateway'ga, aga alates kahest päringust sekundis oli selleks arhitektuuriks Lambda + ALB.

Töö autor loodab, et tulemustest võiks olla abi ka teistele infosüsteemide arhitektidele ja analüütikutele otsuse langetamisel serverivaba arhitektuuri kasutamise osas.

Kasutatud kirjandus

[1] „Status of the CERN httpd“. [WWW] https://www.w3.org/Daemon/ (08.09.2018).

[2] „August 2018 Web Server Survey | Netcraft“. [WWW] https://news.netcraft.com/archives/2018/08/24/august-2018-web-server-survey.html (08.09.2018).

[3] „Announcing Amazon Elastic Compute Cloud (Amazon EC2) - beta“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/ (23.09.2018).

[4] N. Kratzke, „A Brief History of Cloud Application Architectures“, *Applied Sciences*, kd 8, nr 8, lk 1368, aug 2018.

[5] J. Weinman, „Mathematical Proof of the Inevitability of Cloud Computing“, 01.08.2011. [WWW] http://joeweinman.com/Resources/Joe_Weinman_Inevitability_Of_Cloud.pdf (22.09.2016).

[6] G. Adzic ja R. Chatley, „Serverless Computing: Economic and Architectural Impact“, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2017, lk 884–889.

[7] M. Crane ja J. Lin, „An Exploration of Serverless Architectures for Information Retrieval“, *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, New York, NY, USA, 2017, lk 241–244.

[8] M. Villamizar *et al.*, „Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures“, *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, lk 179–182.

[9] A. Eivy, „Be Wary of the Economics of ‘Serverless’ Cloud Computing“, *IEEE Cloud Computing*, kd 4, nr 2, lk 6–12, märts 2017.

[10] M. Malawski, A. Gajek, A. Zima, B. Balis, ja K. Figiela, „Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions“, *Future Generation Computer Systems*, nov 2017.

[11] „AWS Lambda enables functions that can run up to 15 minutes“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/ (25.10.2018).

[12] „What is a Container?“, *Docker*. [WWW] https://www.docker.com/resources/what-container (04.05.2019).

[13] I. Baldini *et al.*, „Serverless Computing: Current Trends and Open Problems“, *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, ja R. Buyya, Toim Singapore: Springer Singapore, 2017, lk 1–20.

[14] „Serverless Architectures“, *martinfowler.com*. [WWW] https://martinfowler.com/articles/serverless.html (13.07.2018).

[15] „Why The Future Of Software And Apps Is Serverless“, *ReadWrite*, 15.10.2012. [WWW] https://readwrite.com/2012/10/15/why-the-future-of-software-and-apps-is-serverless/ (13.11.2018).

[16] „JAWS is now Serverless -- Serverless Code“. [WWW] https://serverlesscode.com/post/serverless-formerly-jaws/ (13.11.2018).

[17] „Google Trends“, *Võrdlus*. [WWW] https://trends.google.com/trends/explore?date=2014-10-01%202018-10-01&q=Serverless,%2Fg%2F11bw4c_dgq (13.11.2018).

[18] A. Williams, *Guide to Serverless Technologies*. The New Stack, 2018.

[19] P. Mell ja T. Grance, „The NIST Definition of Cloud Computing“, National Institute of Standards and Technology, NIST Special Publication (SP) 800-145, sept 2011.

[20] E. van Eyk, A. Iosup, S. Seif, ja M. Thömmes, „The SPEC Cloud Group’s Research Vision on FaaS and Serverless Architectures“, *Proceedings of the 2Nd International Workshop on Serverless Computing*, New York, NY, USA, 2017, lk 1–4.

[21] „The Firebase Blog: Launching Cloud Functions for Firebase v1.0“. [WWW] https://firebase.googleblog.com/2018/04/launching-cloud-functions-for-firebase-1-0.html (02.11.2018).

[22] „docs/introduction.md at master · fnproject/docs · GitHub“. [WWW] https://github.com/fnproject/docs/blob/master/fn/general/introduction.md (22.10.2018).

[23] A. Helendi, „Serverless Survey: +77% Delivery Speed, 4 Dev Workdays/Mo Saved & -26% AWS Monthly Bill“, *Hacker Noon*, 26.04.2018. [WWW] https://hackernoon.com/serverless-survey-77-delivery-speed-4-dev-workdays-mo-saved-26-aws-monthly-bill-d99174f70663 (17.10.2018).

[24] M. Stigler, „Understanding Serverless Computing“, *Beginning Serverless Computing: Developing with Amazon Web Services, Microsoft Azure, and Google Cloud*, M. Stigler, Toim Berkeley, CA: Apress, 2018, lk 1–14.

[25] „UI Testing at Scale with AWS Lambda | AWS DevOps Blog“. [WWW] https://aws.amazon.com/blogs/devops/ui-testing-at-scale-with-aws-lambda/ (18.10.2018).

[26] Y. Cui, „My wish list for AWS Lambda in 2018“, *Binaris Blog*, 18.07.2018. [WWW] https://blog.binaris.com/my-wish-list-for-aws-lambda-in-2018/ (19.10.2018).

[27] B. Kepes, „30% Of Servers Are Sitting ‘Comatose’ According To Research“, *Forbes*. [WWW] https://www.forbes.com/sites/benkepes/2015/06/03/30-of-servers-are-sitting-comatose-according-to-research/ (26.10.2018).

[28] O. Segal, S. Zin, ja A. Shulman, „The Ten Most Critical Security Risks in Serverless Architectures“, 2018. [WWW] https://www.puresec.io/hubfs/SAS-Top10-2018/PureSec%20-%20SAS%20Top%2010%20-%202018.pdf (28.10.2018).

[29] „AWS Lambda – FAQs“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/lambda/faqs/ (28.10.2018).

[30] Y. Cui, „How long does AWS Lambda keep your idle functions around before a cold start?“, *A Cloud Guru*, 04.07.2017. [WWW] https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810 (28.10.2018).

[31] „Best Practices for Working with AWS Lambda Functions - AWS Lambda“. [WWW] https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html (28.10.2018).

[32] S. Can, „Everything you need to know about cold starts in AWS Lambda“, *Hacker Noon*, 01.06.2018. [WWW] https://hackernoon.com/cold-starts-in-aws-lambda-f9e3432adbf0 (28.10.2018).

[33] Y. Cui, „How does language, memory and package size affect cold starts of AWS Lambda?“, *A Cloud Guru*, 15.06.2017. [WWW] https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76 (28.10.2018).

[34] Y. Cui, „I’m afraid you’re thinking about AWS Lambda cold starts all wrong“, *Hacker Noon*, 16.01.2018. [WWW] https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-7d907f278a4f (28.10.2018).

[35] „AWS Lambda Limits - AWS Lambda“. [WWW] https://docs.aws.amazon.com/lambda/latest/dg/limits.html (08.11.2018).

[36] N. Savage, „Going Serverless“, *Commun. ACM*, kd 61, nr 2, lk 15–16, jaan 2018.

[37] „Release: AWS Lambda on 2014-11-13“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/releasenotes/release-aws-lambda-on-2014-11-13/ (19.10.2018).

[38] „AWS Lambda announces service level agreement“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-introduces-service-level-agreement/ (30.10.2018).

[39] „What Is AWS Lambda? - AWS Lambda“. [WWW] https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (04.11.2018).

[40] „Lambda@Edge - AWS Lambda“. [WWW] https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html (04.11.2018).

[41] „New for AWS Lambda – Use Any Programming Language and Share Common Components“, *Amazon Web Services*, 29.11.2018. [WWW] https://aws.amazon.com/blogs/aws/new-for-aws-lambda-use-any-programming-language-and-share-common-components/ (07.05.2019).

[42] „Introducing Azure Functions“. [WWW] https://azure.microsoft.com/en-us/blog/introducing-azure-functions/ (19.10.2018).

[43] Microsoft Azure, „Azure Functions scale and hosting“. [WWW] https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale (07.11.2018).

[44] „Announcing the Azure Functions Premium plan for enterprise serverless workloads“. [WWW] https://azure.microsoft.com/en-us/blog/announcing-the-azure-functions-premium-plan-for-enterprise-serverless-workloads/ (23.04.2019).

[45] R. Meier, „An Annotated History of Google's Cloud Platform“, *Reto Meier*, 10.02.2017. [WWW] https://medium.com/@retomeier/an-annotated-history-of-googles-cloud-platform-90b90f948920 (22.10.2018).

[46] „Cloud Functions serverless platform is generally available“, *Google Cloud Blog*. [WWW] https://cloud.google.com/blog/products/gcp/cloud-functions-serverless-platform-is-generally-available/ (07.11.2018).

[47] „Architect's Recap: IBM Bluemix OpenWhisk“, *IBM Cloud Blog*, 25.02.2016. [WWW] https://www.ibm.com/blogs/bluemix/2016/02/bluemix-openwhisk-overview/ (19.10.2018).

[48] „Elastic Load Balancing pricing - Amazon Web Services“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/elasticloadbalancing/pricing/ (05.05.2019).

[49] „Spotinst Functions“, *Spotinst*. [WWW] https://spotinst.com/products/spotinst-functions/ (22.10.2018).

[50] „Cloudflare Workers“, *Cloudflare*. [WWW] https://www.cloudflare.com/products/cloudflare-workers/ (22.10.2018).

[51] „IronFunctions: the serverless microservices platform by - iron-io/functions“. [WWW] https://github.com/iron-io/functions (03.11.2018).

[52] „Stack Overflow Developer Survey 2018“, *Stack Overflow*. [WWW] https://stackoverflow.com/insights/survey/2018/ (06.11.2018).

[53] „Serverless Docker Beta“. [WWW] https://zeit.co/blog/serverless-docker (06.11.2018).

[54] „4 Misconceptions About Auto Scaling in AWS Cloud Computing“, *Logicworks*, 12.12.2017. [WWW] https://www.logicworks.com/blog/2017/12/common-mistakes-misconceptions-auto-scaling-aws/ (17.11.2018).

[55] „Serverless Framework - Build applications on AWS Lambda, Google CloudFunctions, Azure Functions, AWS Flourish and more“, *serverless*. [WWW] https://serverless.com/framework/ (22.11.2018).

[56] „GitHub - serverless/plugins: Serverless Plugins – Extend the Serverless Framework with these community driven plugins –“. [WWW] https://github.com/serverless/plugins (07.04.2019).

[57] „Key Features of a Content Delivery Network| Performance, Security | Amazon CloudFront“. [WWW] https://aws.amazon.com/cloudfront/features/ (05.05.2019).

[58] „Optimizing AWS Lambda performance with MongoDB Atlas and Node.js“, *MongoDB*. [WWW] https://www.mongodb.com/blog/post/optimizing-aws-lambda-performance-with-mongodb-atlas-and-nodejs (05.04.2019).

[59] „Node.js 8.10 runtime now available in AWS Lambda“, *Amazon Web Services*, 02.04.2018. [WWW] https://aws.amazon.com/blogs/compute/node-js-8-10-runtime-now-available-in-aws-lambda/ (27.02.2019).

[60] Jaak Tepandi, „Tarkvara protsessid, kvaliteet ja standardid“, 03.10.2018. [WWW] http://tepandi.ee/tks-loeng.pdf (06.05.2019).

[61] „8 company changes, millions of answered questions, 1 incredible year“, *SurveyMonkey*. [WWW] https://www.surveymonkey.com/curiosity/2018companyrecap/ (02.03.2019).

[62] M. Clark, „How the BBC builds websites that scale“, *Creative Bloq*. [WWW] https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale (13.04.2019).

[63] „AWS Service Limits - Amazon Web Services“. [WWW] https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html#limits_cloudfront (08.04.2019).

[64] „Apache JMeter - Apache JMeter™“. [WWW] https://jmeter.apache.org/ (01.03.2019).

[65] „JMeterUsers - Jmeter Wiki“. [WWW] https://wiki.apache.org/jmeter/JMeterUsers (04.05.2019).

[66] „EC2 Instance Pricing – Amazon Web Services (AWS)“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/ec2/pricing/on-demand/ (22.04.2019).

[67] „Amazon EBS Pricing - Amazon Web Services“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/ebs/pricing/ (22.04.2019).

[68] „How AWS Pricing Works“, 06.2018. [WWW] https://d1.awsstatic.com/whitepapers/aws_pricing_overview.pdf (21.04.2019).

[69] „AWS Lambda – Pricing“, *Amazon Web Services, Inc.* [WWW] https://aws.amazon.com/lambda/pricing/ (22.04.2019).

[70] „On-Demand vs Reserved vs Spot AWS EC2 Pricing Comparison - BoltOps Blog“. [WWW] https://blog.boltops.com/2018/07/13/on-demand-vs-reserved-vs-spot-aws-ec2-pricing-comparison (04.04.2019).

Lisa 1 – FaaS-teenusplatvormide sisseehitatud päästikud

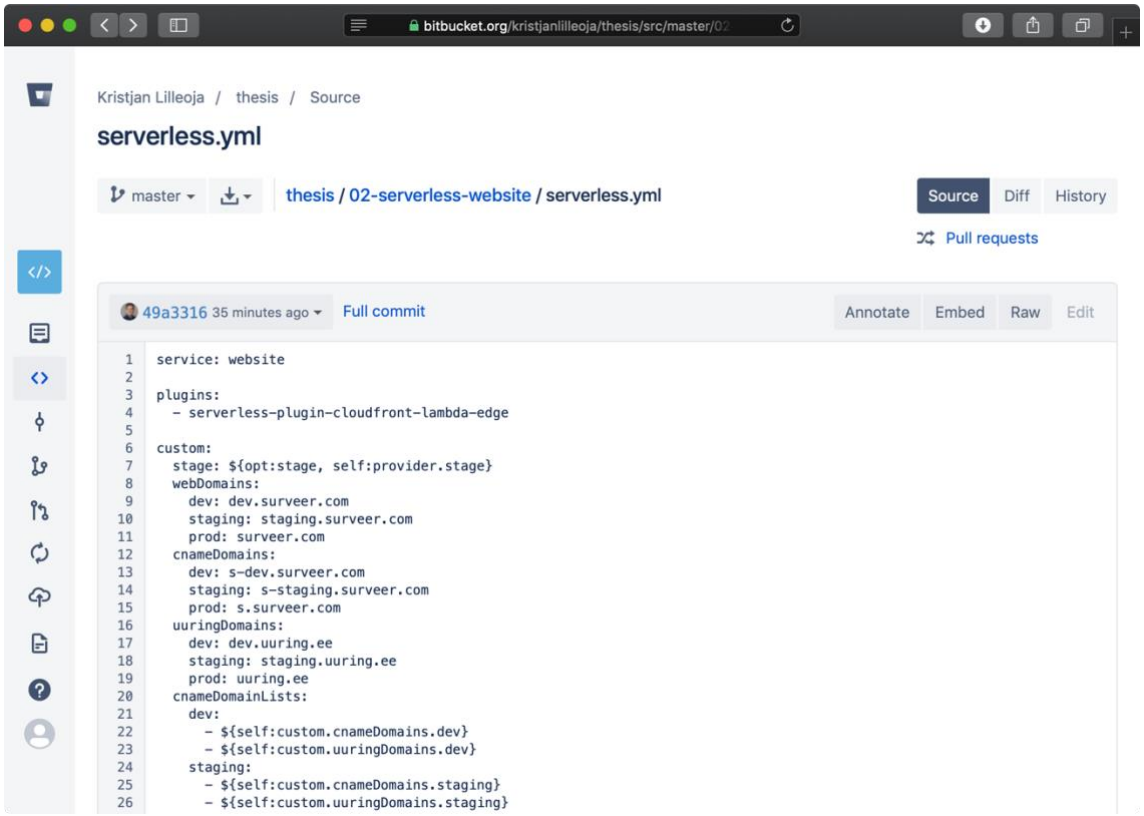
FaaS-teenusplatvormide sisseehitatud päästikud (ingl *triggers*) on erinevad viisid, kuidas funktsioone vastavas platvormis käivitada. Iga platvormi puhul on peamine päästik HTTP/S päring, kuid olenevalt teenusepakkujast on päästikuteks ka hulk teisi teenuseid. AWS'i puhul saab näiteks nii seadistada, et Lambda funktsioon käivitub iga kord, kui uus fail laetakse S3 andmesalvestusteenuse kausta. Järgnevas tabelis on kõik need päästikud peamiste FaaS-teenusplatvormide puhul välja toodud (seisuga 29.04.2019).

AWS Lambda	**Azure Functions**	**Google Cloud Functions**	**IBM Cloud Functions**
Sünkroonsed: Elastic Load Balancing (Application Load Balancer) Amazon Cognito Amazon Lex Amazon Alexa Amazon API Gateway Amazon CloudFront (Lambda@Edge) Amazon Kinesis Data Firehose **Asünkroonsed:** Amazon S3 Amazon SNS Amazon SES AWS CloudFormation Amazon CloudWatch Logs Amazon CloudWatch Events AWS CodeCommit AWS Config **Ajastatud**	HTTP & Webhooks Blob Storage Cosmos DB Event Grid Event Hubs Microsoft Graph Excel tables Microsoft Graph OneDrive files Microsoft Graph Outlook email Microsoft Graph Events Microsoft Graph Auth tokens Mobile Apps Notification Hubs Queue storage SendGrid Service Bus SignalR Table storage Timer Twilio	HTTP Cloud Storage Cloud Pub/Sub Cloud Firestore Firebase (Realtime Database, Storage, Analytics, Auth) Stackdriver Logging	HTTP Alarms Cloudant Message Hub Mobile Push GitHub Custom (hooks, polling, connections) Object Storage events source (experimental)

Lisa 2 – Serverless raamistiku projekti „website“ konfiguratsiooni lähtekood

Surveeri Serverless raamistiku projekti „website“ konfiguratsiooni lähtekood paikneb järgneval aadressil: https://bitbucket.org/kristjanlilleoja/thesis/src/master/02-serverless-website/serverless.yml

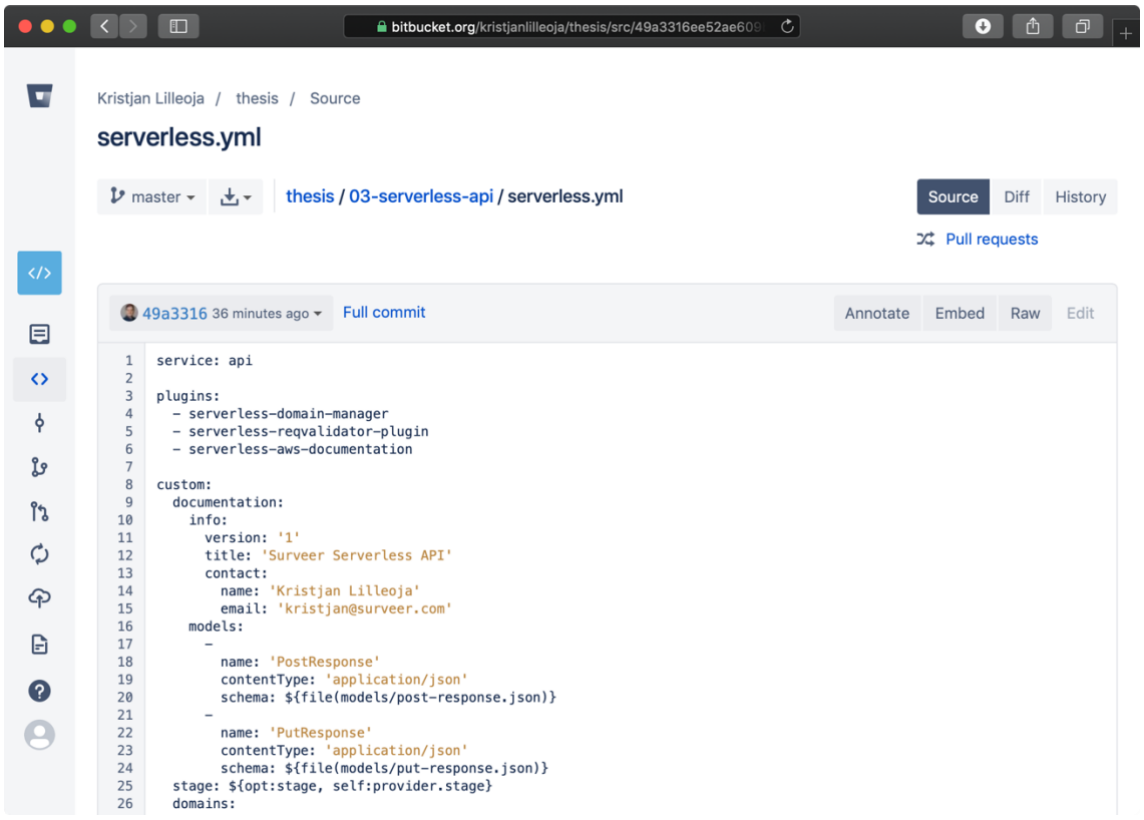
Ekraanivaade kõnealusest failist Bitbucket’is:



Lisa 3 – Serverless raamistiku projekti „api“ konfiguratsiooni lähtekood

Surveeri Serverless raamistiku projekti „api“ konfiguratsiooni lähtekood paikneb järgneval aadressil:
https://bitbucket.org/kristjanlilleoja/thesis/src/master/03-serverless-api/serverless.yml

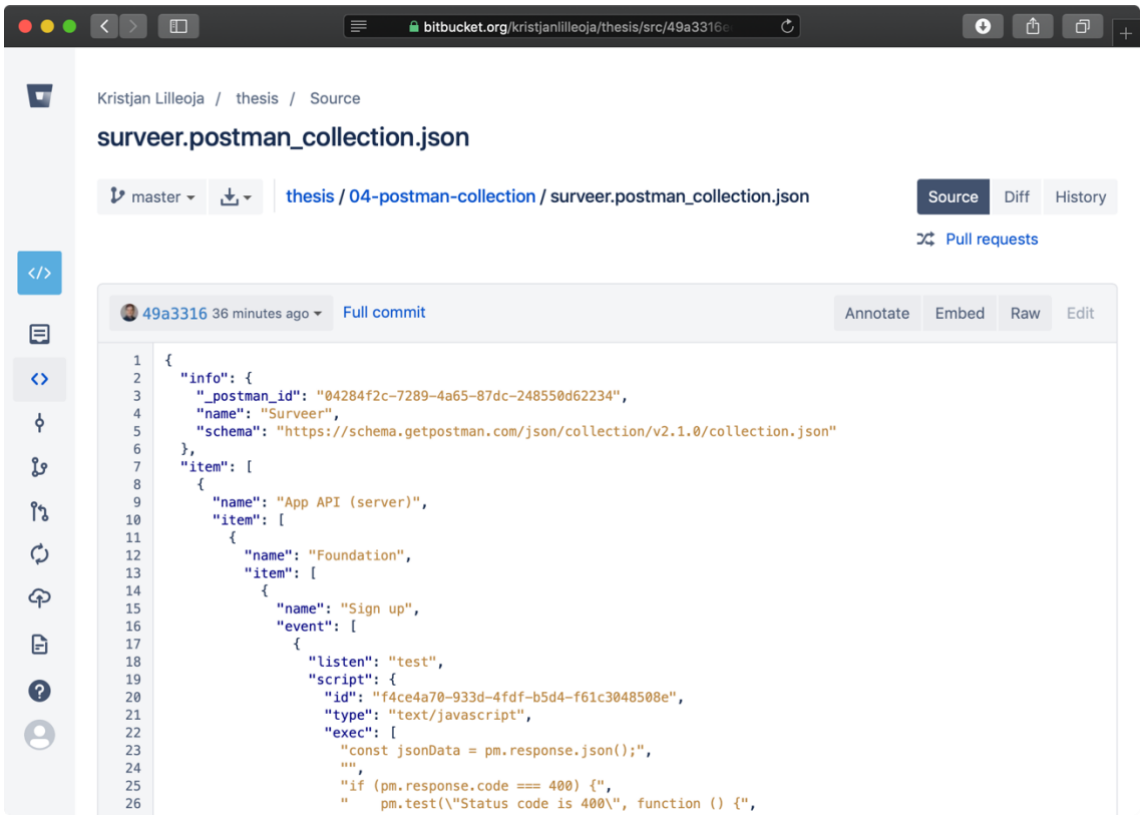
Ekraanivaade kõnealusest failist Bitbucket’is:



Lisa 4 – “Collector API” Postman testide lähtekood

Surveeri “Collector API” Postman testide lähtekood paikneb järgneval aadressil: https://bitbucket.org/kristjanlilleoja/thesis/src/master/04-postman-collection/surveer.postman_collection.json

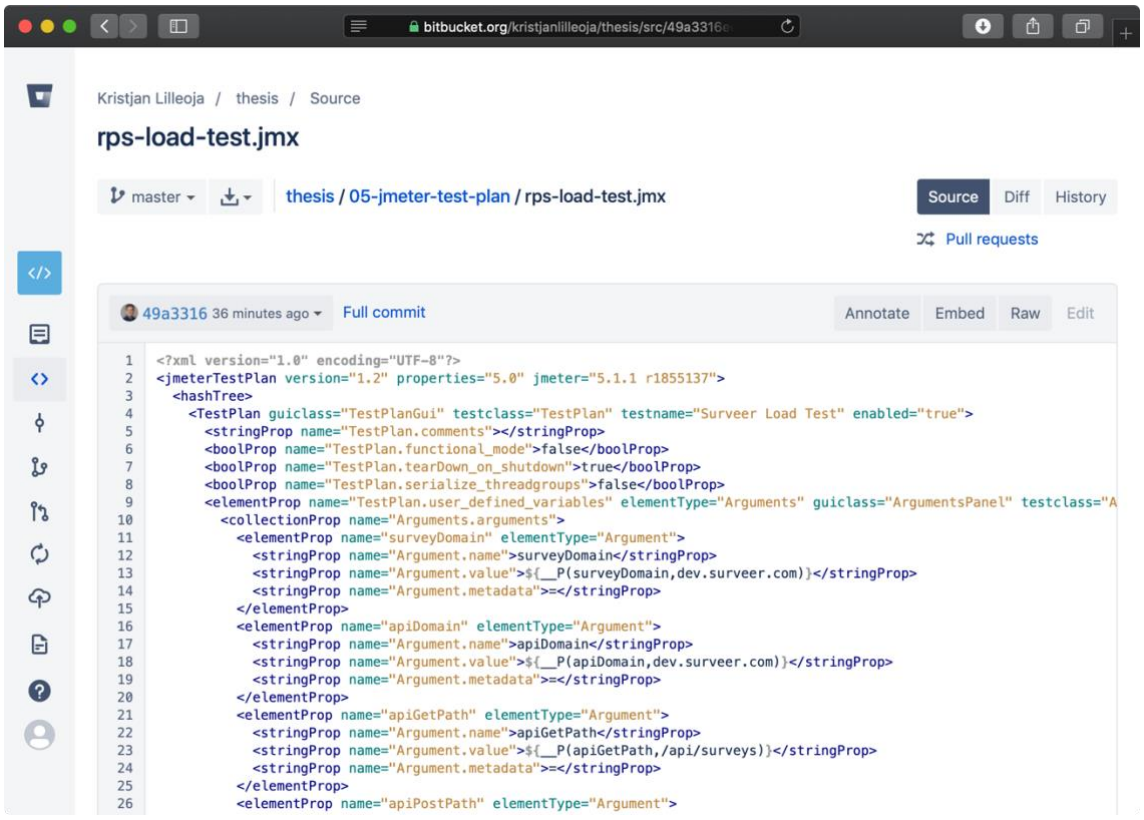
Ekraanivaade kõnealusest failist Bitbucket’is:



Lisa 5 – JMeter testplaani lähtekood

Surveeri JMeter testplaani lähtekood paikneb järgneval aadressil: https://bitbucket.org/kristjanlilleoja/thesis/src/master/05-jmeter-test-plan/rps-load-test.jmx

Ekraanivaade kõnealusest failist Bitbucket'is:

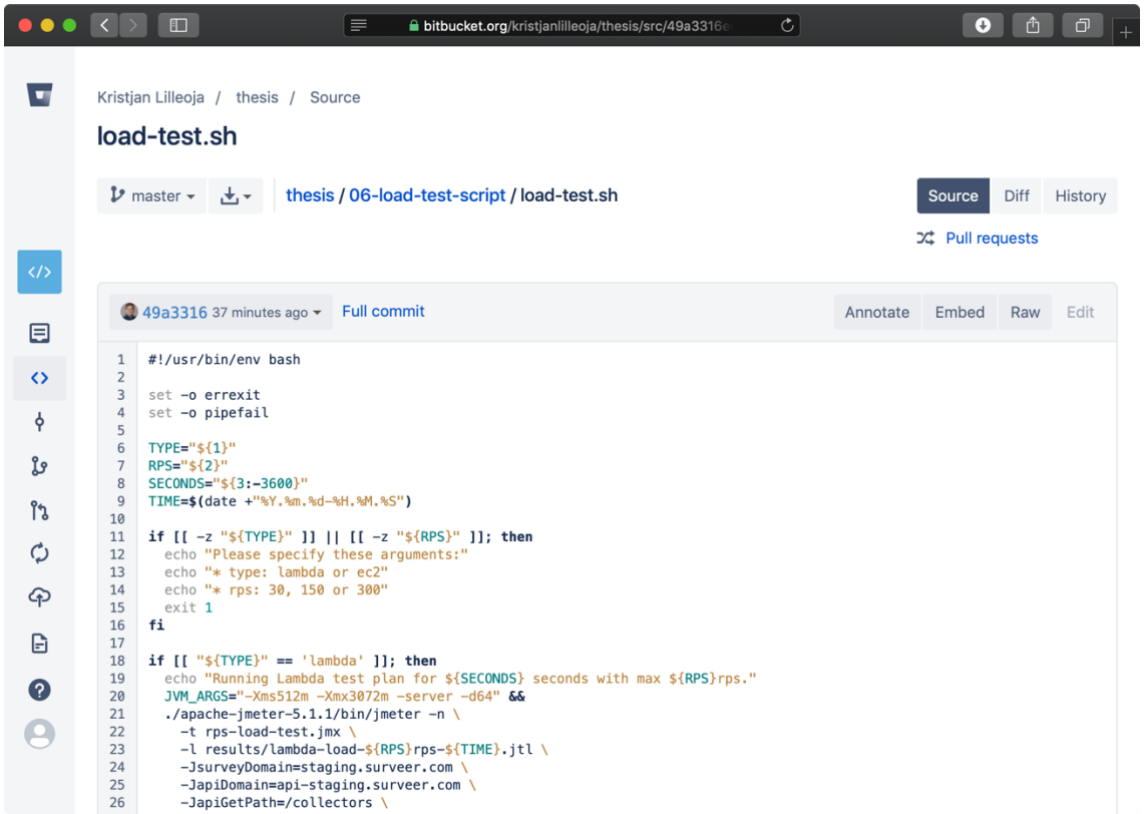


Lisa 6 – Jõudlustestimise *shell*-skripti lähtekood

Jõudlustestimise läbiviimise lihtsustamiseks loodi *shell*-skript, milles paiknes loogika vajalike parameetritega JMeter testplaani käivitamiseks Surveeri varasemal Amazon EC2'el põhineval arhitektuuril või uuel AWS Lambda' põhineval arhitektuuril. Skripti kasutati cron'iga jõudlustestimise ajastamiseks koos kahe sisendparameetriga: tuli sisestada vaid arhitektuur ja päringute arv sekundis. Näiteks EC2 ja stsenaariumi S1 (30 päringut sekundis) puhul oli selleks: „./load-test.sh ec2 30".

Surveeri jõudlustestimise *shell*-skripti lähtekood paikneb järgneval aadressil: https://bitbucket.org/kristjanlilleoja/thesis/src/master/06-load-test-script/load-test.sh

Ekraanivaade kõnealusest failist Bitbucket'is:

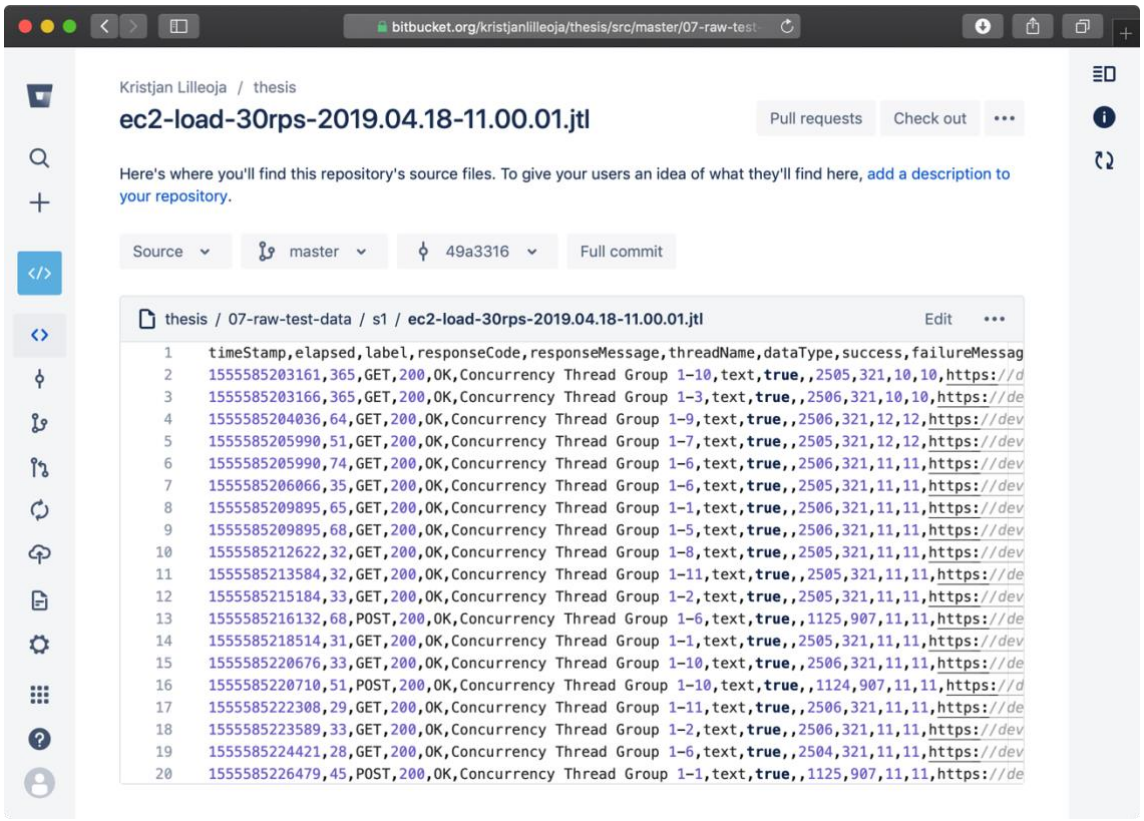


Lisa 7 – Jõudlustestimise tulemuste toorandmed

Surveeri JMeter'iga teostatud jõudlustestimise toorandmed on talletatud .jtl laiendiga failidesse (mis on sisuliselt CSV-tüüpi tekstifailid). Neid faile on kokku 18, igas failis on talletatud kõik testplaani 60 minuti jooksul tehtud päringud ja nendega seotud metaandmed. Need on jaotatud iga stsenaariumi kohta (S1, S2, S3), mille puhul tehti mõlema arhitektuuriga (EC2, Lambda) kolm tunniajast testseeriat.

Surveeri jõudlustestimise toorandmed (kokku ligi 1 GB) paiknevad järgneval aadressil: https://bitbucket.org/kristjanlilleoja/thesis/src/master/07-raw-test-data/

Ekraanivaade kõnealuste tulemuste esimesest failist Bitbucket'is:

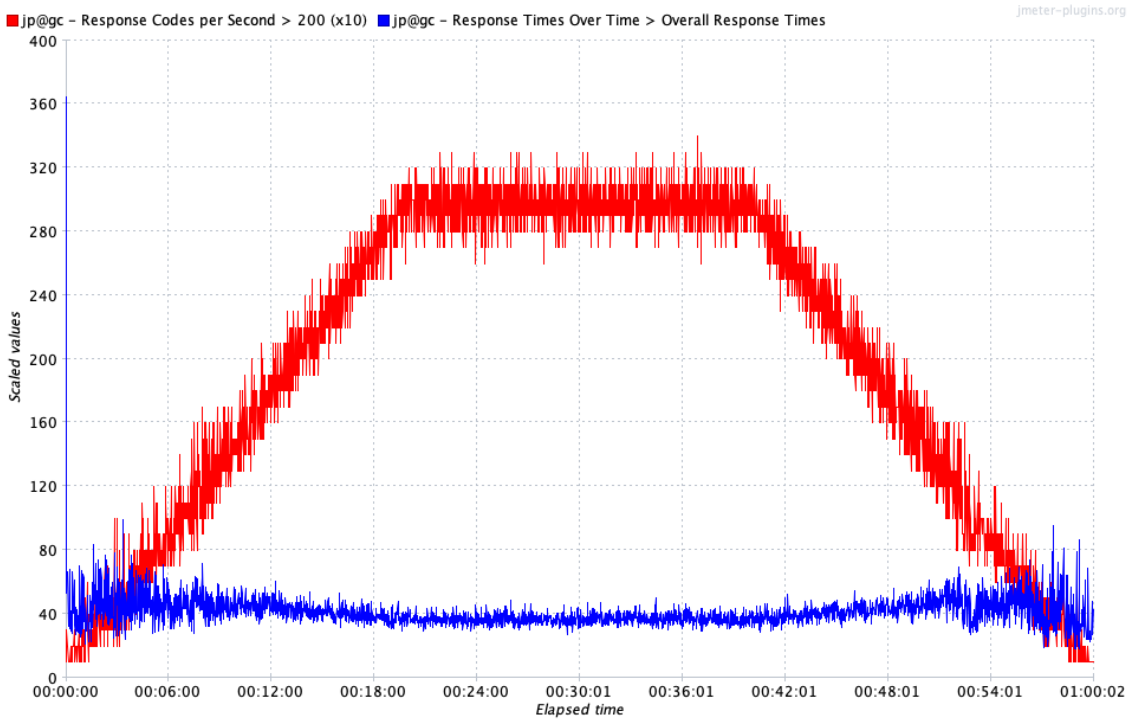


Lisa 8 – Stsenaariumi S1 tulemuste graafikud

Käesolevas lisas on esitatud stsenaariumi S1 (tippkoormusega 30 API-päringut sekundis) jõudlustestimise tulemuste graafikud. Esimesed kolm graafikut kujutavad Surveeri varasema arhitektuuri (EC2 + ALB) ja viimased kolm uue serverivaba arhitektuuri (Lambda + API Gateway) testimistulemusi. Graafikutel on kujutatud õnnestunud päringute arv sekundis (päringud HTTP-oleku koodidega 200 ja 204) ja päringute keskmine reaktsiooniaeg. Selleks, et andmete korrelatsioon esile tuleks, on need ühistele telgedele paigutatud. Seda on tehtud õnnestunud päringute arvu sekundis andmete korrutamisega, mis tähendab, et andmete lugemisel on oluline jälgida konkreetse graafiku andmete korrutise numbrit (nt esimesel graafikul on see 10x).

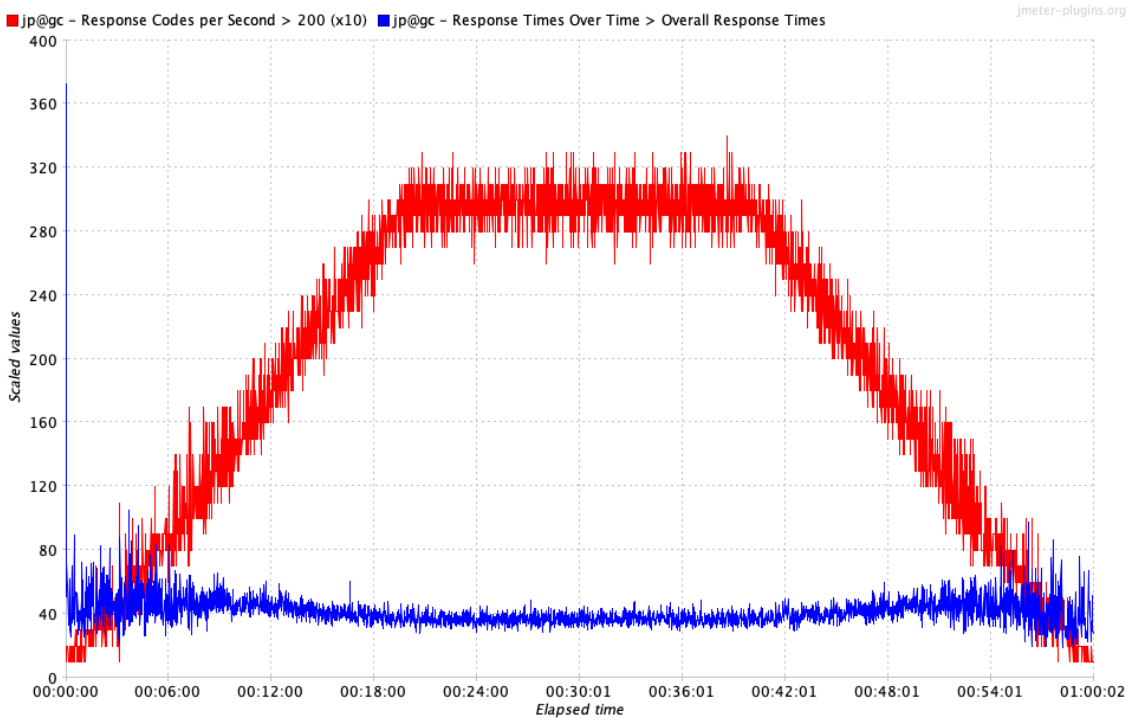
S1 – 1. EC2 testimine (ec2-load-30rps-2019.04.18-11.00.01.jtl)

Graafikult on näha, et EC2 + ALB arhitektuuri testimise päris alguses olid keskmisest suuremad reaktsiooniajad (üle 360 ms), kuid üldise testimise lõikes mingeid probleeme täheldada pole, koormuse kasvades jõudlus ühtlustus.



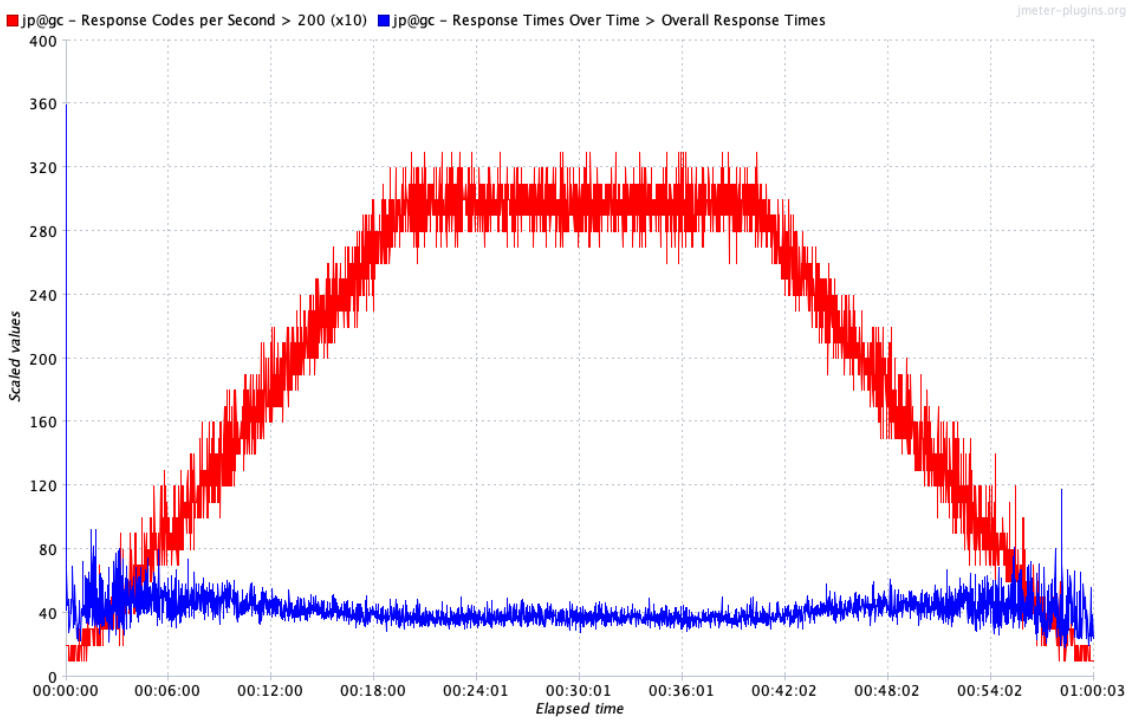
S1 – 2. EC2 testimine (ec2-load-30rps-2019.04.18-13.00.01.jtl)

Graafikult on näha, et sarnaselt eelmise testimisega oli EC2 + ALB arhitektuuri testimise päris alguses keskmisest suuremad reaktsiooniajad (üle 360 ms), kuid üldise testimise lõikes mingeid probleeme täheldada pole, koormuse kasvades jõudlus ühtlustus.



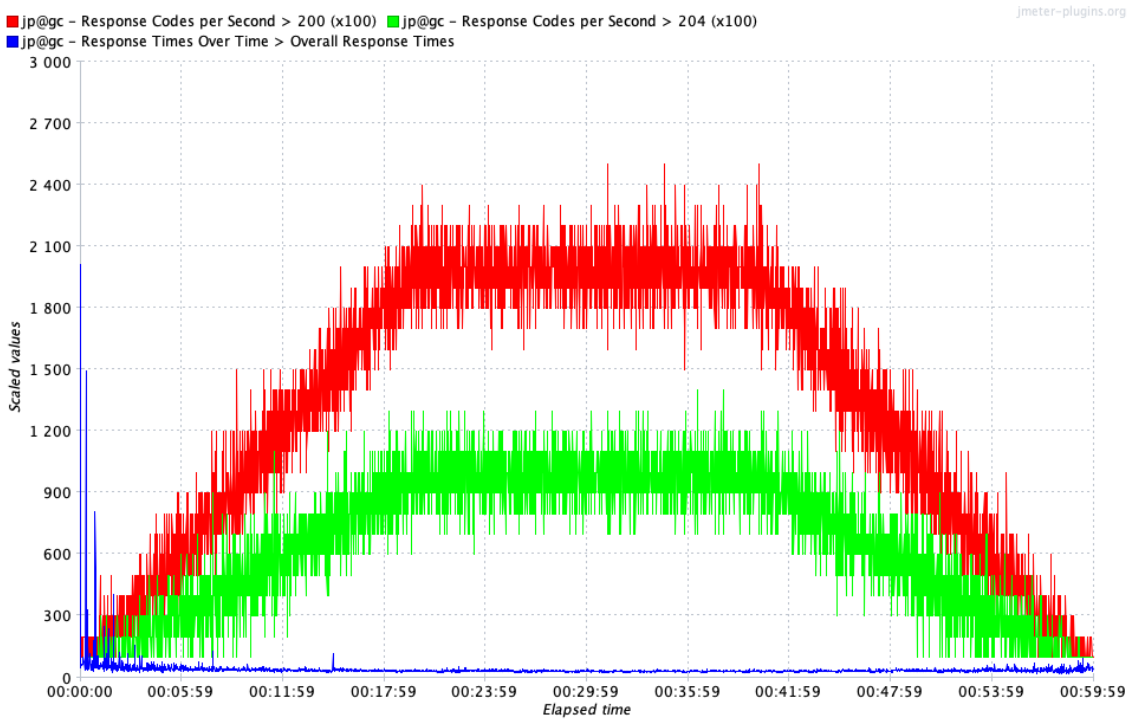
S1 – 3. EC2 testimine (ec2-load-30rps-2019.04.18-15.00.01.jtl)

Graafikult on näha, et EC2 + ALB arhitektuuri testimise päris alguses olid keskmisest suuremad reaktsiooniajad (360 ms piirimail), kuid üldise testimise lõikes mingeid probleeme täheldada pole, pigem on märgata sarnaselt eelmiste testide tulemustele, et koormuse kasvades jõudlus ühtlustus.



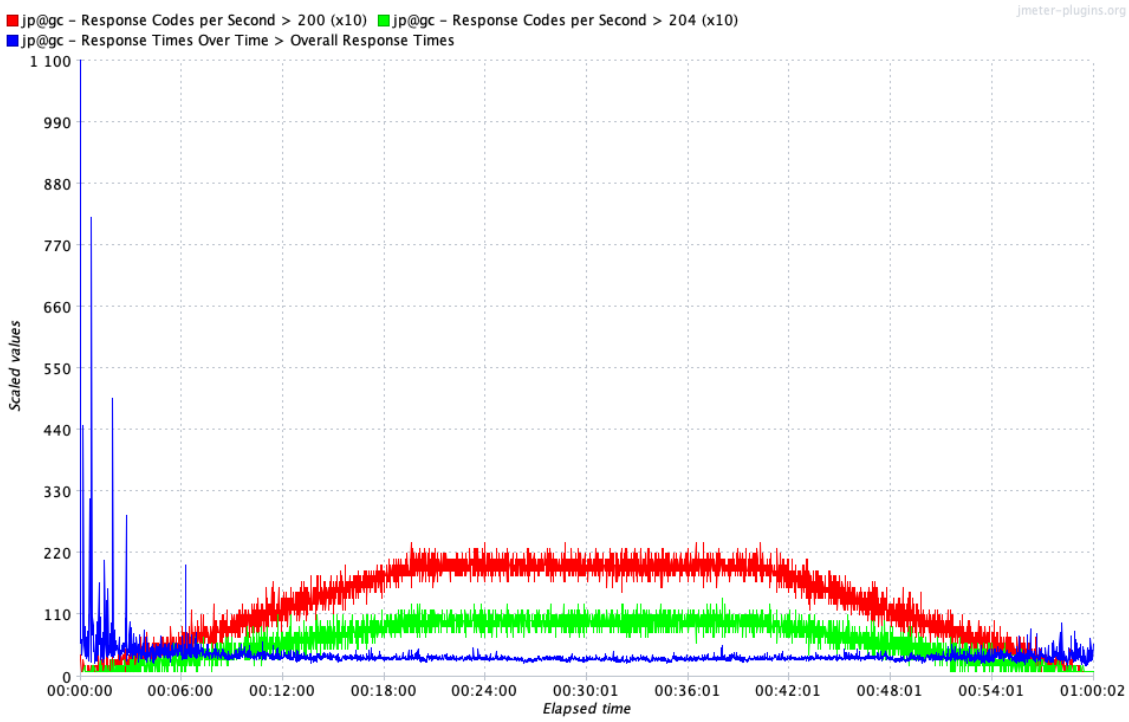
S1 – 1. Lambda testimine (lambda-load-30rps-2019.04.18-12.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (kuni 2000 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi, mõlemad vastavad oodatavale trendile.



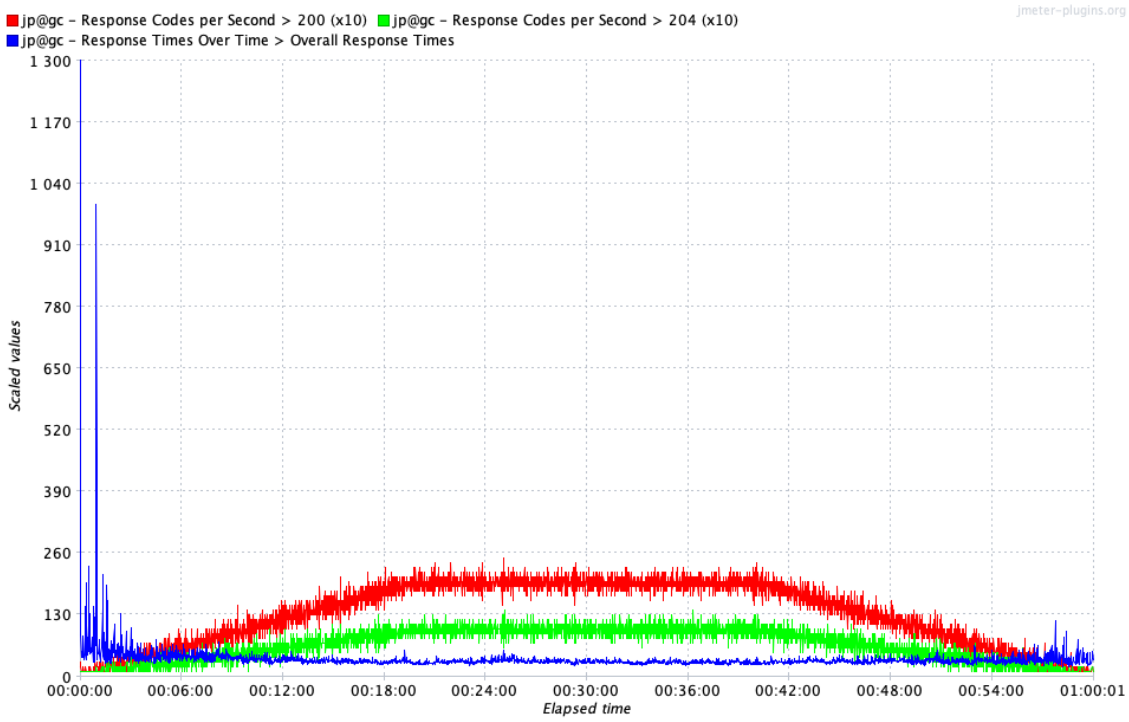
S1 – 2. Lambda testimine (lambda-load-30rps-2019.04.18-14.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (kuni 1100 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Võrreldes eelmise testimisega olid külmkäivitused lühemad. Päringute õnnestumised vastavad oodatavale trendile.



S1 – 3. Lambda testimine (lambda-load-30rps-2019.04.18-16.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (üle 1300 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Reaktsiooniajad olid sarnased eelmise testimisega. Päringute õnnestumised vastavad oodatavale trendile.

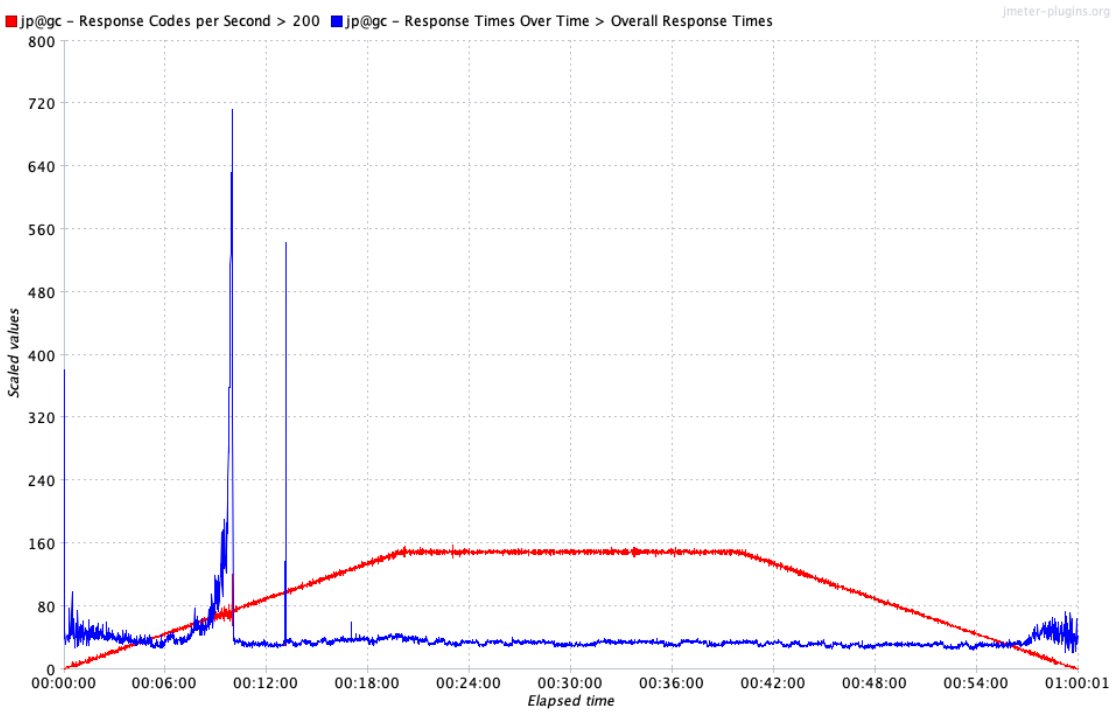


Lisa 9 – Stsenaariumi S2 tulemuste graafikud

Käesolevas lisas on esitatud stsenaariumi S2 (tippkoormusega 150 API-päringut sekundis) jõudlustestimise tulemuste graafikud. Esimesed kolm graafikut kujutavad Surveeri varasema arhitektuuri (EC2 + ALB) ja viimased kolm uue serverivaba arhitektuuri (Lambda + API Gateway) testimistulemusi. Graafikutel on kujutatud õnnestunud päringute arv sekundis (päringud HTTP-oleku koodidega 200 ja 204) ja päringute keskmine reaktsiooniaeg. Selleks, et andmete korrelatsioon esile tuleks, on need ühistele telgedele paigutatud. Seda on tehtud õnnestunud päringute arvu sekundis andmete korrutamisega, mis tähendab, et andmete lugemisel on oluline jälgida konkreetse graafiku andmete korrutise numbrit (nt esimesel graafikul see puudub).

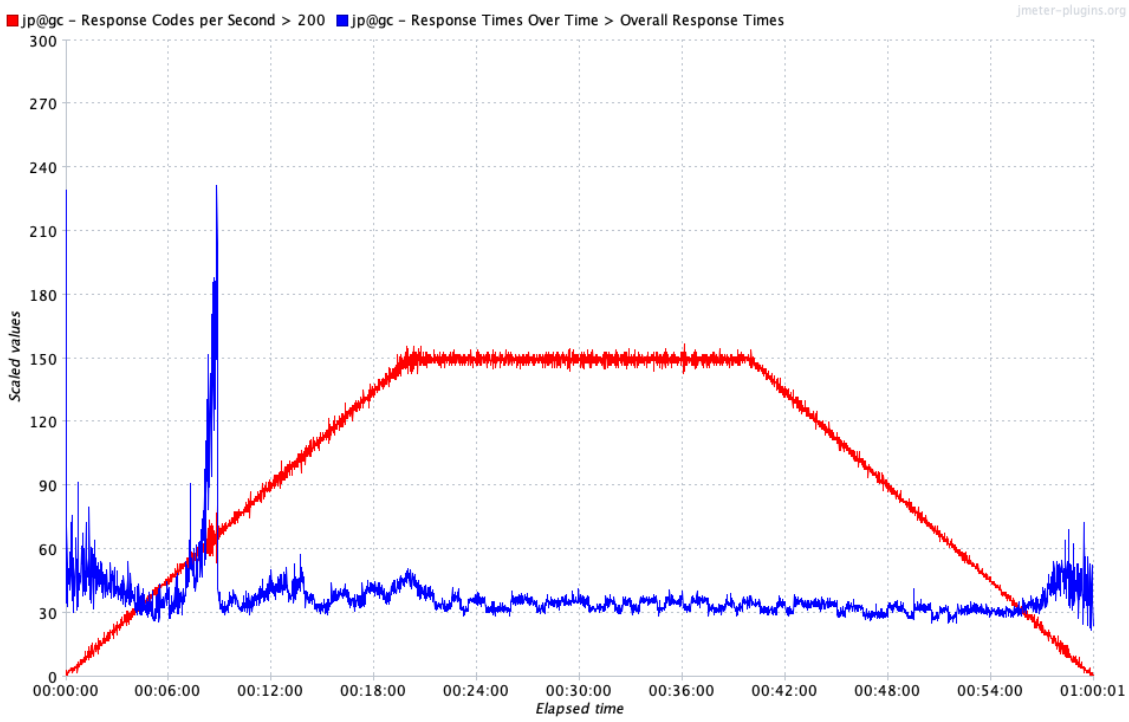
S2 – EC2 1. testimine (ec2-load-150rps-2019.04.18-18.00.01.jtl)

Graafikult on näha, et lisaks EC2 + ALB arhitektuuri testimise algusele kasvas reaktsiooniaeg hüppeliselt umbes kaheksandal minutil (kuni 720 ms). See on juba märk sellest, et tõenäoliselt ei tulnud automaatne skaleerimine EC2 instantside lisamisega piisavalt kiiresti toime. Samas mingit ilmselgelt päringute ebaõnnestumist ei toimunud.



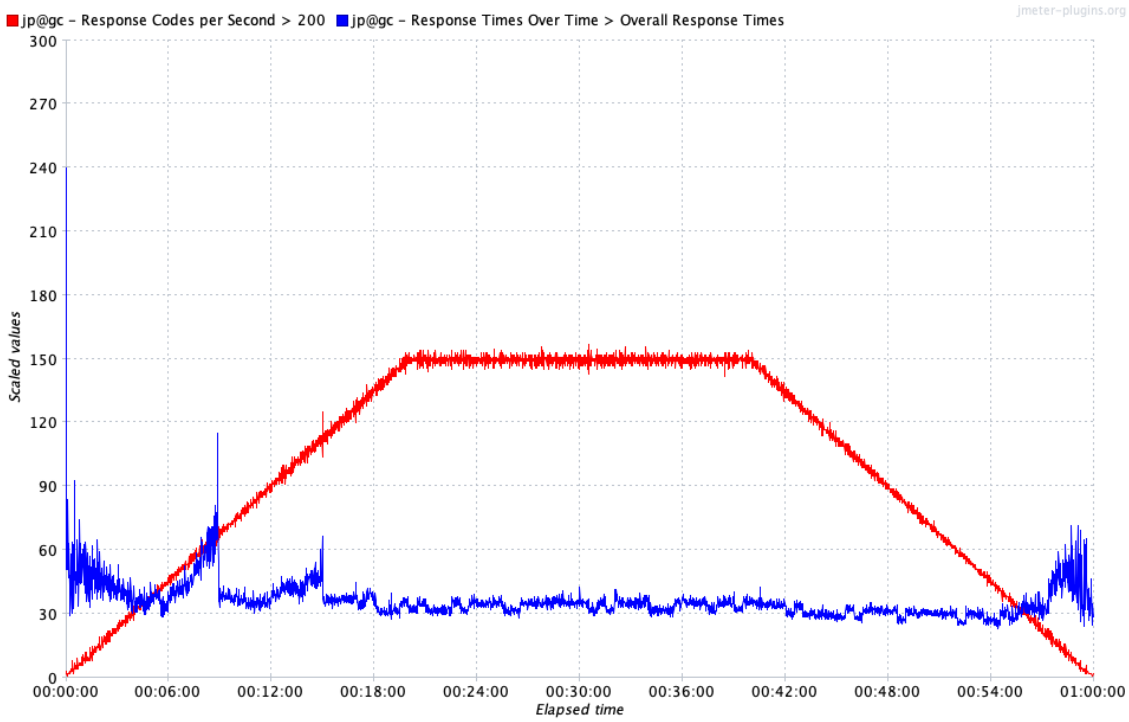
S2 – EC2 2. testimine (ec2-load-150rps-2019.04.18-20.00.01.jtl)

Graafikult on näha, et lisaks EC2 + ALB arhitektuuri testimise algusele kasvas reaktsiooniaeg hüppeliselt umbes kaheksandal minutil (kuni 230 ms). Sarnaselt eelmisele testimisele on märgata automaatse skaleerimisega seotud probleeme, kuid samas mingit ilmselgelt päringute ebaõnnestumist ei toimunud.



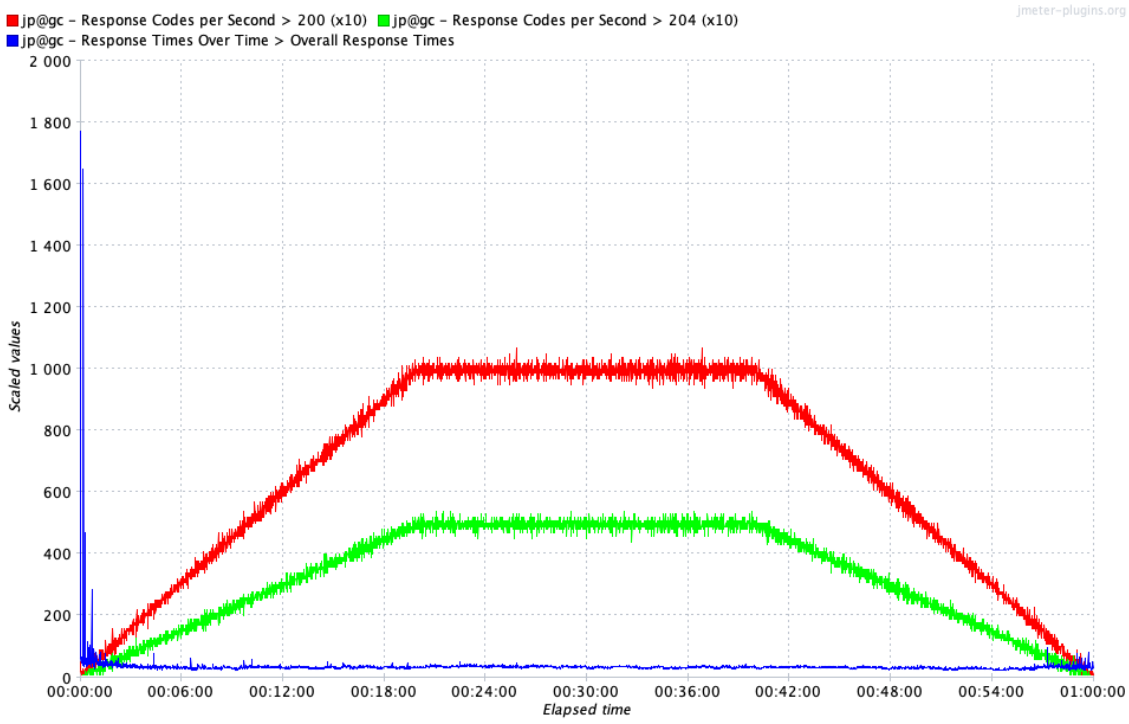
S2 – EC2 3. testimine (ec2-load-150rps-2019.04.18-22.00.01.jtl)

Graafikult on näha, et lisaks EC2 + ALB arhitektuuri testimise algusele kasvas reaktsiooniaeg umbes kaheksandal minutil (kuni 120 ms). Võrreldes eelmiste testimistega näitas antud testimine veidi paremaid tulemusi.



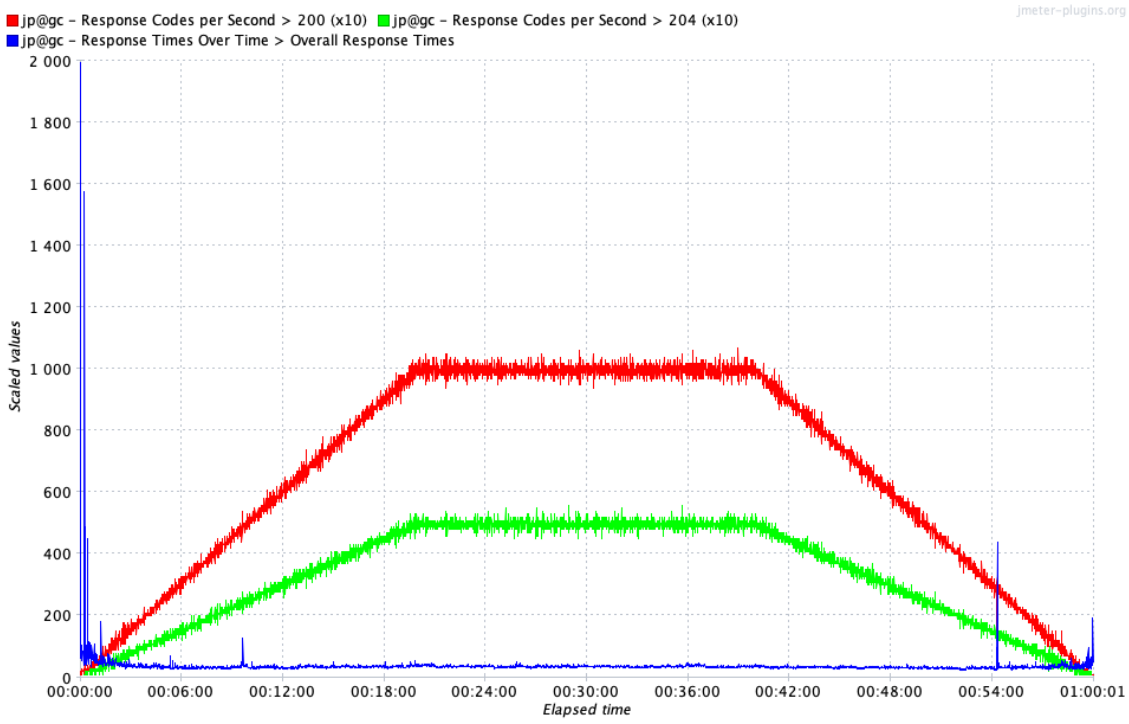
S2 – Lambda 1. testimine (lambda-load-150rps-2019.04.18-19.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (kuni 1800 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi, mõlemad vastavad oodatavale trendile. Erinevalt EC2'e testimistulemustele, koormuse kasvades üllatusi ei tekkinud.



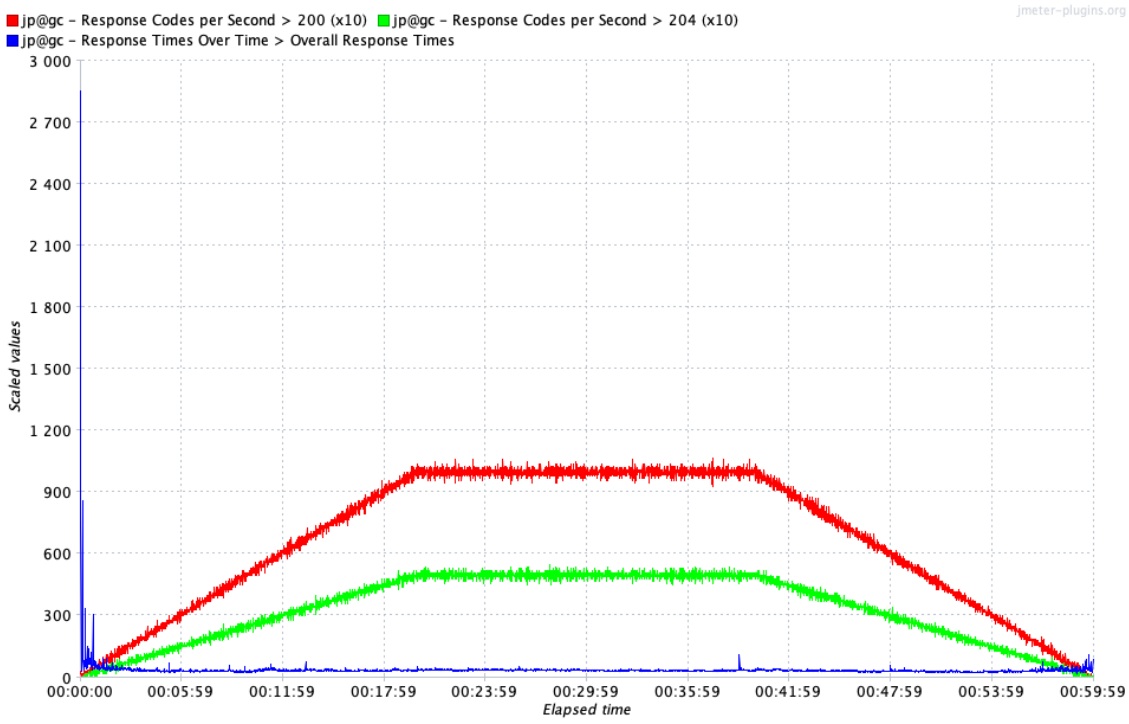
S2 – Lambda 2. testimine (lambda-load-150rps-2019.04.18-21.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (isegi üle 2000 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi, mõlemad vastavad oodatavale trendile. Graafikult hakkab silma väike reaktsiooniaja hüpe 54ndal minutil pisut üle 400 ms, muus osas üllatusi ei esinenud.



S2 – Lambda 3. testimine (lambda-load-150rps-2019.04.18-23.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (isegi üle 2800 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi, mõlemad vastavad oodatavale trendile. Üldiselt graafikul üllatusi ei esinenud.

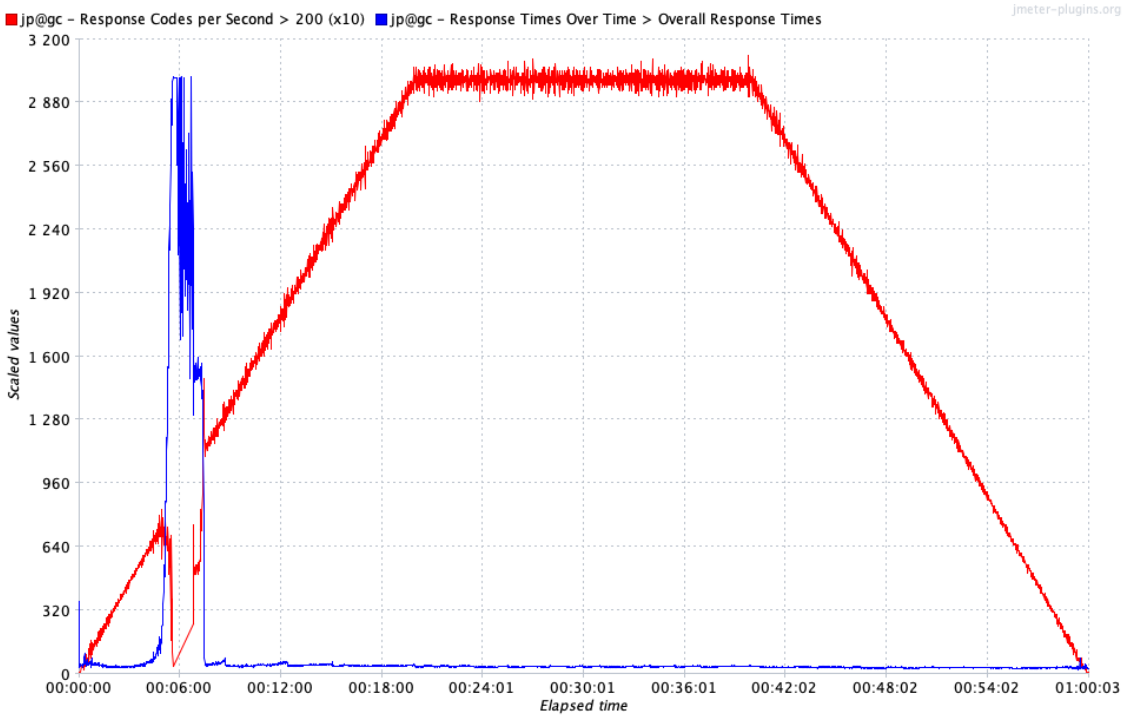


Lisa 10 – Stsenaariumi S3 tulemuste graafikud

Käesolevas lisas on esitatud stsenaariumi S3 (tippkoormusega 300 API-päringut sekundis) jõudlustestimise tulemuste graafikud. Esimesed kolm graafikut kujutavad Surveeri varasema arhitektuuri (EC2 + ALB) ja viimased kolm uue serverivaba arhitektuuri (Lambda + API Gateway) testimistulemusi. Graafikutel on kujutatud õnnestunud päringute arv sekundis (päringud HTTP-oleku koodidega 200 ja 204) ja päringute keskmine reaktsiooniaeg. Selleks, et andmete korrelatsioon esile tuleks, on need ühistele telgedele paigutatud. Seda on tehtud õnnestunud päringute arvu sekundis andmete korrutamisega, mis tähendab, et andmete lugemisel on oluline jälgida konkreetse graafiku andmete korrutise numbrit (nt esimesel graafikul on see 10x).

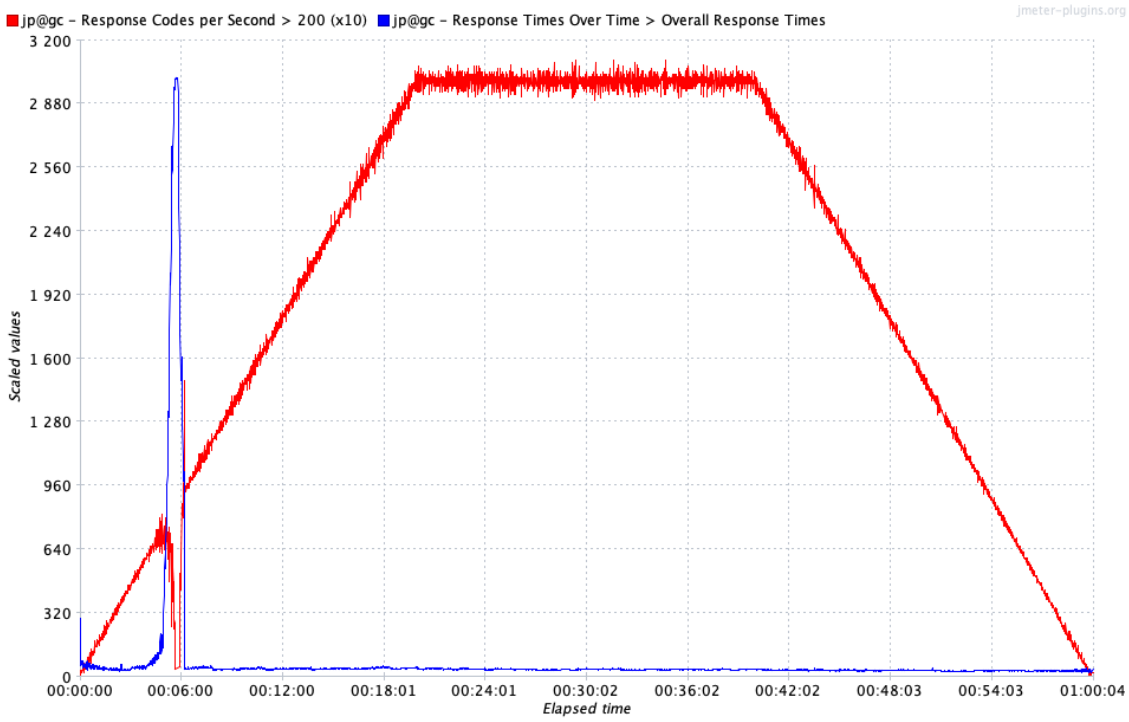
S3 – 1. EC2 testimine (ec2-load-300rps-2019.04.19-03.00.01.jtl)

Graafikult on näha, et EC2 + ALB arhitektuuri testimise ajal tekkis ilmselge päringute ebaõnnestumine enne viiendat minutit ja kestis umbes neli minutit. See on näide sellest, kui koormus kasvab kiiremini, kui automaatne skaleerimine jõuab uusi EC2 instantse lisada.



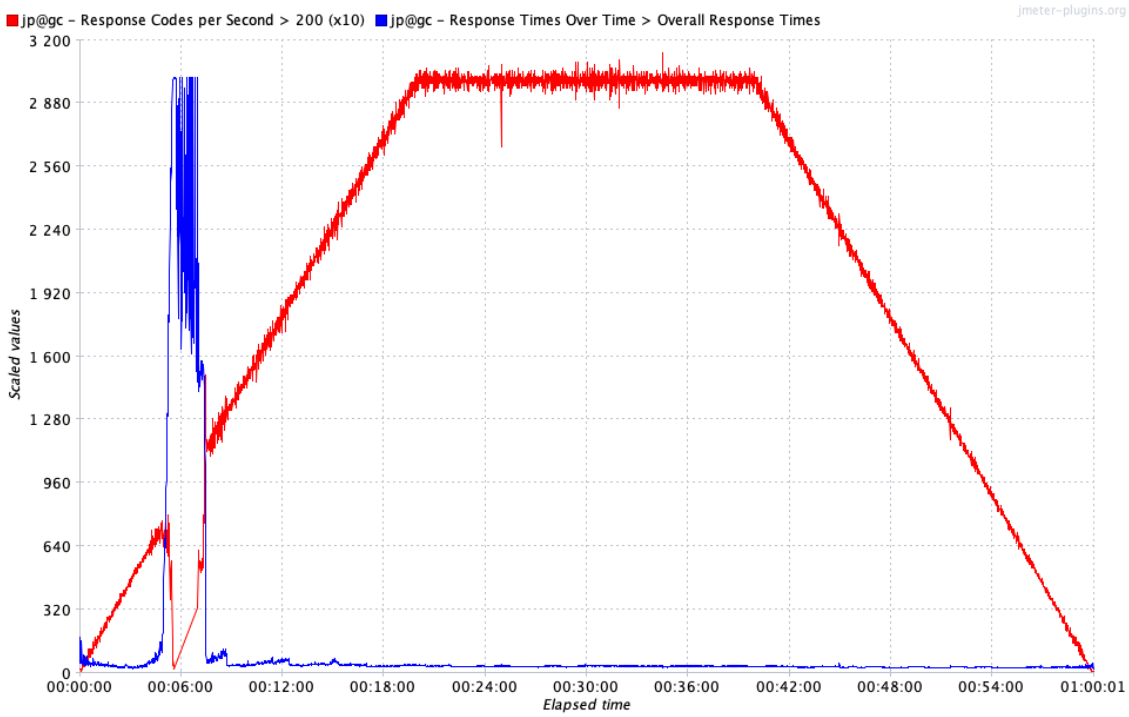
S3 – 2. EC2 testimine (ec2-load-300rps-2019.04.19-05.00.01.jtl)

Graafikult on näha, et EC2 + ALB arhitektuuri testimise ajal tekkis ilmselge päringute ebaõnnestumine umbes viiendal minutil. See on väga sarnane eelmise testimise tulemustega, aga problemaatiline periood oli antud juhul lühem (umbes 2 minutit). Sarnaselt eelmise testimise tulemustele ei esinenud pärast seda perioodi probleeme.



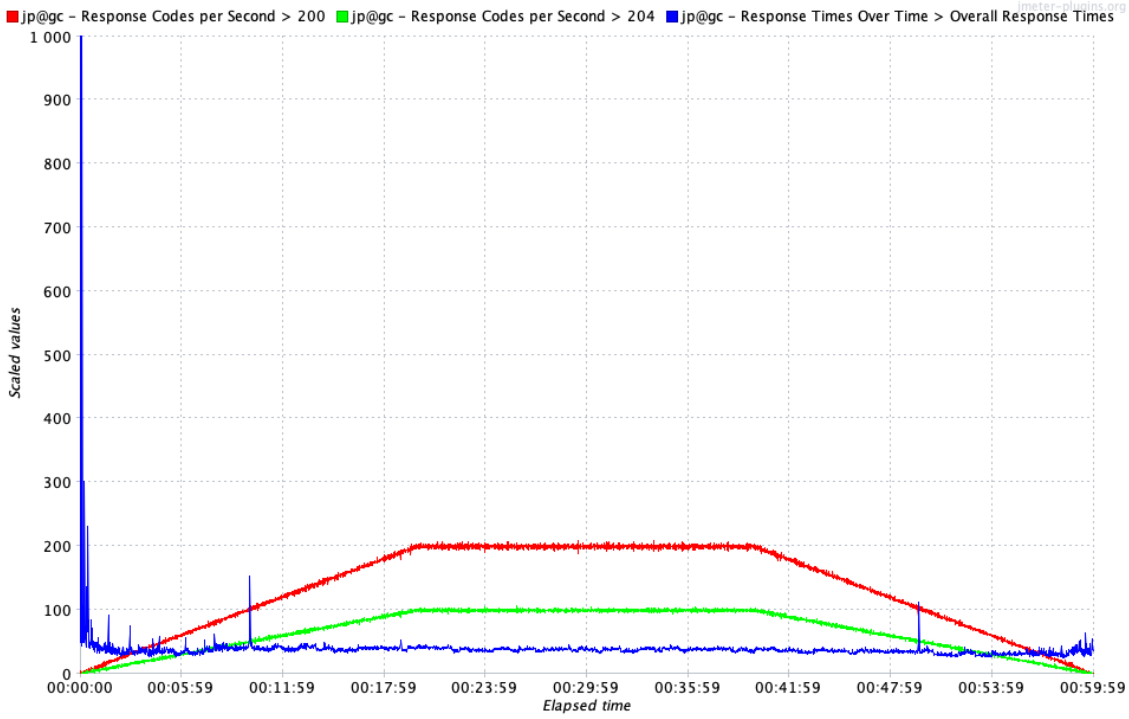
S3 – 3. EC2 testimine (ec2-load-300rps-2019.05.06-20.00.01)

Graafikult on näha, et EC2 + ALB arhitektuuri testimise ajal tekkis ilmselge päringute ebaõnnestumine sarnaselt eelmise testimisega umbes viiendal minutil ja kestis umbes kolm minutit. Sarnaselt eelmise testimise tulemustele ei esinenud pärast seda perioodi probleeme.



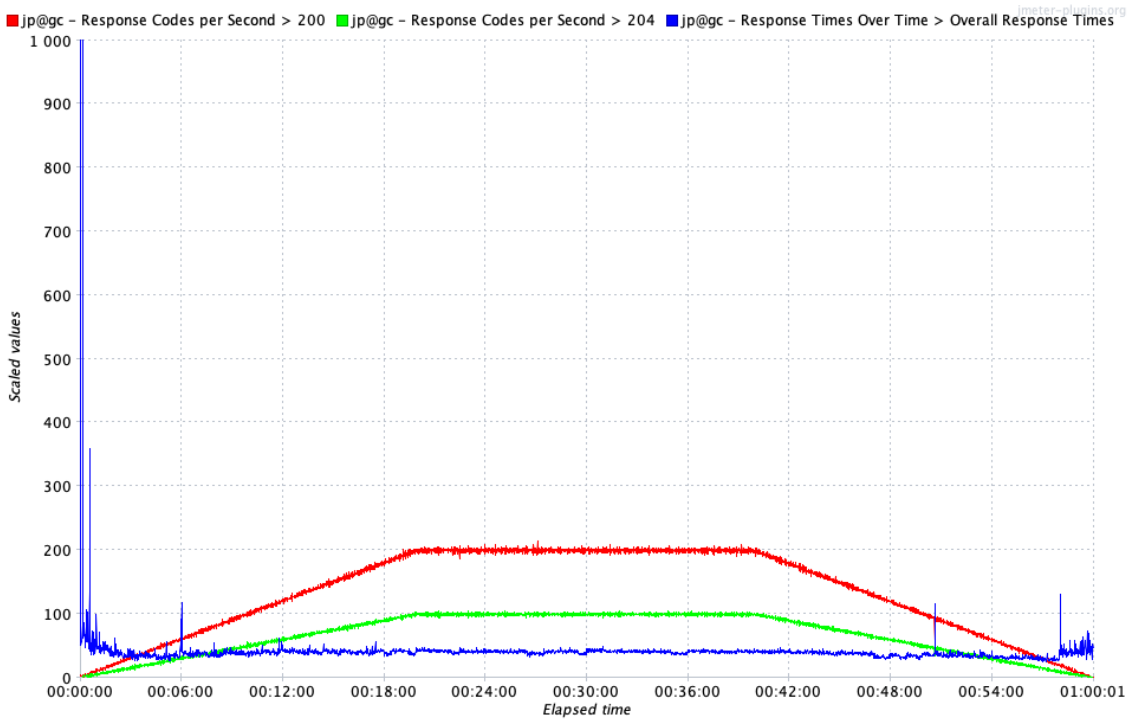
S3 – 1. Lambda testimine (lambda-load-300rps-2019.04.19-04.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (üle 1000 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi, mõlemad vastavad oodatavale trendile. Esines küll väiksemaid reaktsiooniaja hüppeid (nt 9. ja 50. minuti kandis), kuid erinevalt EC2'e testimistulemustele koormuse kasvades erilisi üllatusi ei tekkinud.



S3 – 2. Lambda testimine (lambda-load-300rps-2019.04.19-06.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (üle 1000 ms) – see on FaaS-funktsioonidele omane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi, mõlemad vastavad oodatavale trendile. Tulemused olid väga sarnased eelmistele testimistulemustele.



S3 – 3. Lambda testimine (lambda-load-300rps-2019.04.19-12.00.01.jtl)

Graafikult on näha, et Lambda + API Gateway arhitektuuri testimise alguses oli mitmel hetkel keskmisest suuremaid reaktsiooniaegu (üle 1000 ms) – see on FaaS-funktsioonidele omapärane külmkäivitus. Kuna uus POST-otspunkt vastab HTTP-oleku 200 asemel 204-le, siis ka graafikul on kujutatud GET (punane) ja POST (roheline) päringute õnnestumised eraldi. Suurem erinevus võrreldes eelmiste testimistega oli see, et umbes 49. minutil on märgata õnnestunud päringute kukkumist eeldatavast trendist. Samas ei väljendu see probleem reaktsiooniaegades, mis olid tol hetkel igati korras.