TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Anton Budovey 232825IAIB
Artjom Hruljov 232990IAIB

Kooud ja Arete teenuste migreerimine skaleeruvale arhitektuurile seire toega

Bakalaureusetöö

Juhendaja: Bahdan Yanovich
BSc

Tallinn 2026

Autorideklaratsioon

Kinnitame, et oleme 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.

Autorid: Anton Budovey ja Artjom Hruljov

12.06.2026

Lühikokkuvõte

Käesolev bakalaureusetöö käsitleb Kooud ja Arete tarkvarasüsteemide operatsioonilisi kitsaskohti ning pakub lahendused kolmes valdkonnas: logimine, seire ja infrastruktuur.

Bakalaureusetöö klient on Tallinna Tehnikaülikooli tarkvaraarendaja Maksim Usmanov, kelle vastutusalasse kuulub Kooud ja Arete uuendamise protsess ja vastavate süsteemide administreerimine.

Töö analüüsiosas tuvastatakse süsteemi peamised probleemid - ühe tõrkepunkti risk, puuduv püsisalvestus, ressursside ületarbimine ning ebaühtlane ja keskse lahenduseta logimine. Teoreetilise alusena käsitletakse vaadeldavuse kontseptsiooni ning Kubernetese kui konteinerorkestratsioonisüsteemi põhimõtteid.

Metoodikas analüüsitakse turul saadaolevaid seire- ja orkestratsiooniplatvormi alternatiive ning põhjendatakse valitud lahendusi. Seirelahendusena valitakse Grafana Stack, mis koosneb Grafanast, Lokist ja Mimirist. Konteinerorkestratsiooni platvormina valitakse Kubernetes.

Lahenduse arenduse käigus töötatakse välja ühtne logimisstruktuur, mis rakendatakse kõigis põhiteenustes. Logimissüsteem toetab nelja logimistaset, logide puhverdamist, lokaalset varundamist ning teenustevahelist päringu jälgimist unikaalse identifikaatori abil. Juurutatakse Grafana Stack, mis võimaldab koondada kõigi teenuste logid ja mõõdikud ühte kohta. Teenused migreeritakse Kubernetese süsteemile, kus rakendatakse automaatne taastumine, automaatne taastumine ning RabbitMQ püsisalvestus ning automaatne juurutamine pideva lõimimise ja tarnimise torustiku kaudu.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 52 leheküljel, 24 peatükki, 12 joonist, 3 tabelit.

Abstract

Migration of Kooud and Arete services to scalable architecture with monitoring support

This bachelor's thesis addresses the operational shortcomings of the Kooud and Arete software systems and proposes solutions in three areas: logging, monitoring, and infrastructure.

The client of the bachelor's thesis is Maksim Usmanov, a software engineer at Tallinn University of Technology, whose area of responsibility includes planning and development of Kooud and Arete software as well as administration of used servers and workloads.

The analysis section identifies the main problems of the system - single point of failure risk, lack of persistent storage, resource overconsumption, and inconsistent and non-centralised logging. The theoretical foundation covers the concept of observability and the principles of Kubernetes as a container orchestration system.

The methodology analyses available monitoring and orchestration platform alternatives and justifies the chosen solutions. Grafana Stack, consisting of Grafana, Loki, and Mimir, is selected as the monitoring solution. Kubernetes is selected as the container orchestration system.

During development, a unified logging was implemented across all core services, supporting four log levels, buffering, local backup, and cross-service request tracing. Grafana Stack centralised logs and metrics, while migration to Kubernetes enabled automatic recovery, RabbitMQ persistent storage, and automated deployments.

The thesis is in Estonian and contains 52 pages of text, 24 chapters, 12 figures, 3 tables.

Lühendite ja mõistete sõnastik

API	Rakendusliides (*Application Programming Interface*)
AWS	Amazoni veebiteenused (*Amazon Web Services*)
CI/CD	Pidev lõimimine ja tarnimine (*Continuous Integration / Continuous Delivery*)
CPU	Keskseade (*Central Processing Unit*)
DNS	Domeeninimi süsteem (*Domain Name System*)
DOKS	DigitalOceani Kubernetese teenus (*DigitalOcean Kubernetes Service*)
HPA	Horisontaalne automaatne skaleerimine (*Horizontal Pod Autoscaler*)
HTTP	Andmevahetusprotokoll (*Hypertext Transfer Protocol*)
HTTPS	Turvaline andmevahetusprotokoll (*Hypertext Transfer Protocol Secure*)
JSON	Andmevahetuse vorming (*JavaScript Object Notation*)
LogQL	Loki päringukkeel (*Log Query Language*)
OS	Operatsiooni süsteem (*Operation System*)
PromQL	Prometheuse päringukkeel (*Prometheus Query Language*)
PVC	Püsiva salvestusmahu taotlus (*PersistentVolumeClaim*)
RAM	Mälu (*Random Access Memory*)
RFC	*Request for Comments*
S3	Lihtne salvestusteenus (*Simple Storage Service*)
SQL	Struktureeritud päringukkeel (*Structured Query Language*)
SSH	Turvaline kaugühenduse protokoll (*Secure Shell*)
TPC-B	Tehingute töötlemise jõudluse standard (*Transaction Processing Performance Council Benchmark B*)
URL	Ühtne ressursi aadress (*Uniform Resource Locator*)
VPN	Virtuaalne privaatvõrk (*Virtual Private Network*)
VU	Virtuaalne kasutaja (*Virtual User*)
YAML	Andmete serialiseerimise vorming (*YAML Ain't Markup Language*)

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Kooud (varem tuntud kui Charon) ja Arete on TalTechis arendatud tarkvarasüsteem, mis toetab programmeerimisainete õppetööd. See loob keskkonna, kus üliõpilased saavad esitada oma koodilahendusi ning õppejõud neid käsitsi või automaatselt kontrollida ja hinnata [1] [2]. Hetkel on Kooud arendus- ja migratsioonifaasis ning ei ole veel tootmiskeskkonnas kasutusel. Käesolev bakalaureusetöö kuulub Kooudi migratsiooni ja arenduse protsessi ning keskendub süsteemi tehnilisele täiustamisele.

Täna on Kooud ja Arete süsteemid ebastabiilsed ega suuda toime tulla suurenenud koormusega, eriti eksamite ajal, kui süsteem sageli kokku variseb ning taastamine võtab aega, mille jooksul õpilased ei saa töid esitada. Arendajatel puudub ülevaade süsteemi toimimisest, mistõttu vigade otsimine ja arendus on aeglasem.

Kuigi Aretes on skaleerimisvõimekus arhitektuuriliselt ette nähtud, ei toeta seda praegune infrastruktuur, kuna kõik teenused töötavad ühel serveril ning nende käsitsi haldamine on keeruline ja ajamahukas.

Lisaks on seirelahendus ebaühtlane: iga teenus kasutab oma logimisviisi, logid on ebastandardsed ning mõõdikute vaatamine ei ole kõigi teenuste puhul saadaval ja kasutajaliides on ebamugav, kuna puudub sorteerimine ja filtreerimine. Samuti puudub keskne vaade serveri koormusele, kettaruumi täitumisele, turvariskidele ja logidele.

Töö eesmärk on lahendada Kooudi ja Arete süsteemi peamised operatsioonilised kitsaskohad. Selle saavutamiseks on seatud alameesmärgid:

- Ühtlustada logimine kõikides teenustes, ehk rakendada ühtse logimisstandardi.
- Luua keskne seire- ja analüüsilahendus, ehk juurutada süsteemi, mis kogub logi- ja mõõdikuandmeid.
- Migreerida Kooud ja Arete teenused orkestreerimissüsteemile, et tagada süsteemi kõrgem töökindlus, skaleeritavus ja lihtsustada infrastruktuuri haldust.

2 Metoodika

Käesolevas peatükis kirjeldatakse lõputöö raames kasutatud metoodikat ja tööprotsessi, mis on vajalik Kooud ja arete süsteemide moderniseerimiseks. Peatükis antakse esmalt põhjalik ülevaade uuritavast objektist ehk Kooud ja Arete tarkvarasüsteemide praegusest tehnilisest seisukorrast ja tuvastatud kriitilistest probleemidest. Seejärel kirjeldatakse tööetappe, mis võimaldavad tuvastatud probleemid lahendada.

2.1 Praegune Kooud seisund

Kooud ja Arete on Tallinna Tehnikaülikoolis arendatud tarkvara, mis koosneb viiest teenusest: Moodle'i plugin, Kooud tagarakendus, *Request Forwarder*, *Test Runner* ja *tester*. Süsteem hõlmab ka andmebaase ja sõnumivahendajat. Kuigi süsteemi arhitektuur on skaleerimiseks ette nähtud, ei suuda praegune ühe serveri infrastruktuur kasvavat koormust toetada - esituste hulga suurenemine koormab süsteemi üha rohkem, mõjutades otseselt teenuste töökindlust.

Programmeerimisülesannete testimise voog illustreerib neid probleeme kõige selgemalt. Kui tudeng esitab lahenduse, liigub see läbi mitme teenuse: kui tudeng esitab lahenduse, võtab Kooud Backend esituse vastu, seostab selle tudengi ja kursuse andmetega ning edastab *Request Forwardile*. *Request Forwarder* suunab esituse RabbitMQ sõnumijärjekorda, kust *Test Runner* selle loeb ja käivitab vastava testija konteineri. Testimise tulemus saadetakse tagasi *Request Forwardile*, mis edastab selle Kooud tagarakendusele hinde kirjutamiseks. Lisas 2 olev joonis illustreerib programmeerimisülesannete testimise voogu.

Iga esituse jaoks käivitab *Test Runner* eraldi Docker konteineri, mis sisaldab vastava programmeerimiskeele testimiskeskkonna. Süsteem toetab mitmeid erinevaid testimiskeskkondi (näiteks Python, Python Notebook, Java, F# ja Perl), millest igaüks on loodud ja hooldatud õppeassitentide ning arendajate poolt. Selleks pääseb *Test Runner* ligi serveri Docker daemonile otse läbi `/var/run/docker.sock` ja töötab privilegeeritud režiimis. See kuju-

tab endast tõsist turvaprobleemi: privilegeeritud konteineris töötav viga või pahatahtlik tudengikood võib potentsiaalselt saada kontrolli kogu serveri üle. Kuna kõik teenused töötavad samal serveril, tähendaks serveri kompromiteerimine ligipääsu nii andmebaasidele, tudengite koodile kui ka kogu süsteemi infrastruktuurile.

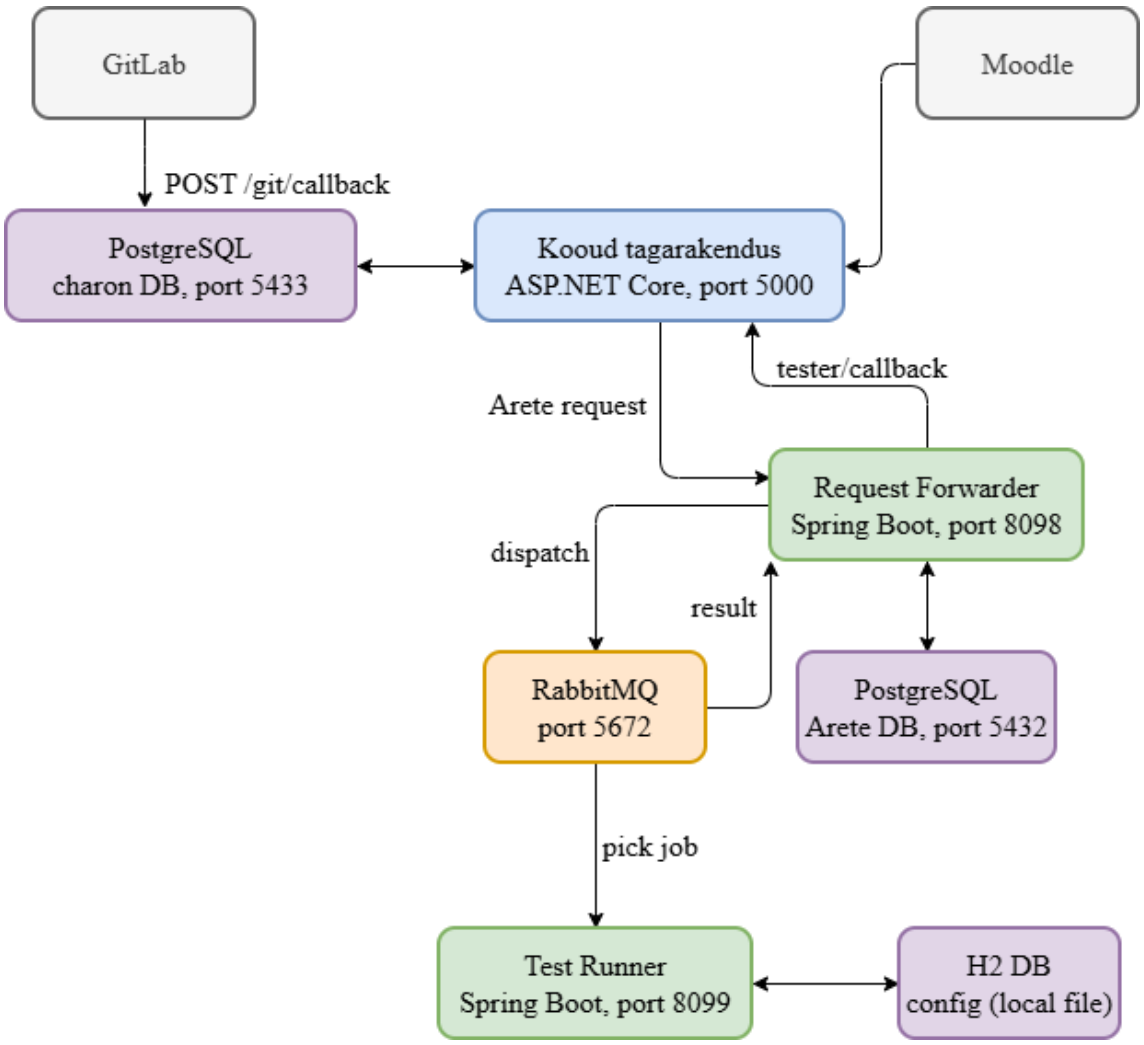
Ressursikasutuse osas esineb süsteemil mitmeid kriitilisi kitsaskohti. Kõik teenused - *Request Forwarder*, *Test Runner* ja *testerid* - töötavad ühel serveril, mistõttu konteinerpiltide suurus, mis ulatub 500 megabaidist kuni 2 gigabaidini, täidab kettaruumi kiiresti. Lisaks keerukamad testid, näiteks masinõppe mudeleid käivitavad testerid, genereerivad suuri ajutisi artefakte, mis salvestatakse samale serverile. Samaaegselt töötavad testerid tarbivad märkimisväärselt muutmälu ning on esinenud intsidente, kus muutmälu täielik täitumine takistas uute testerite käivitamist. See on otseselt koormuse tasakaalustamise probleem - mitme vähem võimsa serveri vahel koormuse jagamine vähendaks üksiku serveri ülekoormust ning parandaks süsteemi töökindlust koormuse tipphetkedel.

Sõnumivahendaja tasandil on esinenud kriitilisi intsidente, kus RabbitMQ sõnumijärjekorda saadeti koos päringuga ka selle metaandmed ja failid, mille kogusmaht ületas RabbitMQ limiidi. Selle tulemusena lõpetas *Request Forwarder* päringute saatmise ning süsteemi taastamiseks tuli käsitsi serverisse siseneda, taaskäivitada RabbitMQ ja seejärel *Request Forwarder*. Kõige kriitilisem tagajärg oli see, et kõik järjekorras olnud esitused kadusid jäädavalt - puuduva püsisalvestuse tõttu ei taastunud ükski neist. Sarnane probleem esineb ka *Test Runneri* puhul: kui see katkeb pärast esituste järjekorrast lugemist, kaovad kõik aktiivsed esitused taaskäivituse käigus.

Infrastruktuuri tasandil kujutab ühe serveri kasutamine tootmiskeskkonnas tõsist riski. On esinenud intsidente, kus tulemüüri vale seadistus põhjustas ligipääsu kaotuse serverile ning teenuse taastamine võttis 12-14 tundi. Puuduvad varukoopiad ning tsentraliseeritud seire, mistõttu peavad arendajad probleemide tuvastamiseks käsitsi iga konteineri logisid kontrollima.

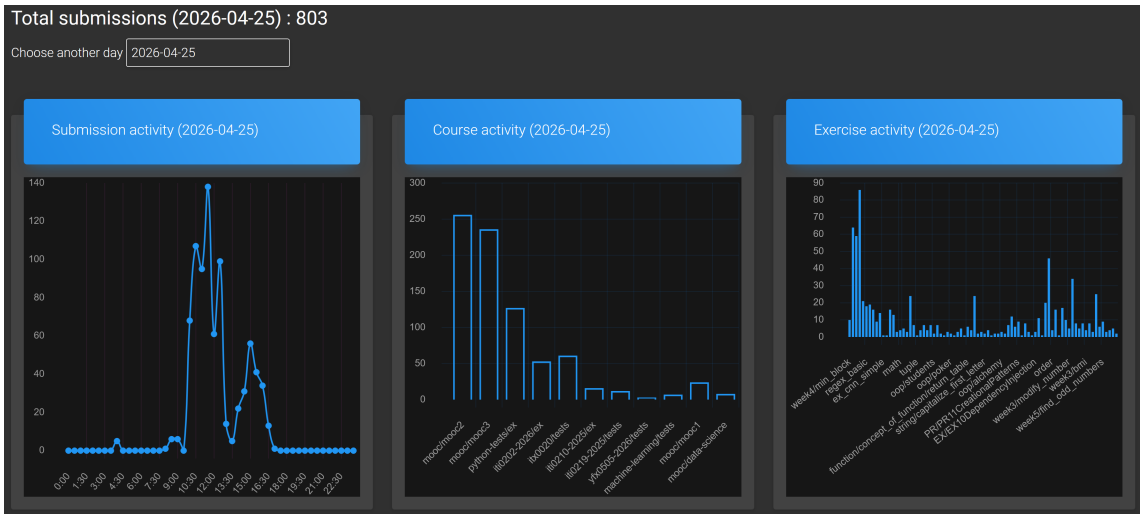
Nagu joonisel 1 näha, on Kooud väga keerukas ning ühelgi neist teenustest puudub seirefunktsiooni, välja arvatud *Request Forwarder*. *Request Forwarderil* on oma seirelahendus, mis ei kasuta ühtegi valmis tööriista: teenus saadab API (*Application Programming Interface*) kaudu eesrakendusele oma riistvara kasutuse andmeid, nagu CPU (*Central Processing*

Unit) ja RAM (*Random Access Memory*) kasutus protsentides ning kasutatud kettaruumi maht. Eesrakendus kuvab need andmed graafikutena (vt joonis 2).



Joonis 1. Praegune Kooud ja Arete arhitektuur.

Nagu joonis 2 näitab, see töölaud ei paku paindlikku filtreerimist ega sorteerimist ning võimaldab andmeid filtreerida ainult päeva kaupa. Lisaks on olemas logide töölaud, kus arendajad näevad ainult infot tehtud esitamiste kohta, kuid sügavamaks analüüsiks peavad nad pöörduma Docker konteinerite süsteemilogide poole. Olukord on eriti halb ülejäänud teenustes, kus puudub igasugune logide ja mõõdikute koondamine ning arendajad on sunnitud probleemide tuvastamiseks kontrollima iga Docker konteineri andmeid eraldi.



Joonis 2. Eksisteeriv töölaud.

2.2 Töö plaan

Töö käigus analüüsime esmalt olemasolevaid kitsaskohti ning võrdleme turul olevaid seire- ja orkestratsiooniplatvorme. Analüüsi tulemuste põhjal koostame ühtse logimisstandardi, mis kirjeldab logide vormingut, tasemeid ja nõutud välju, ning teostame vajalikud muudatused teenuste koodibaasides. Paralleelselt kavandame Kubernetese klastri arhitektuuri, määratledes nimeruumid, teenuste omavahelised sõltuvused ning andmebaaside ja sõnumivahendaja konfiguratsioonid. Seejärel juurutame valitud seirelahenduse ning migreerime teenused Kubernetese platvormile, alustades konfiguratsioonide ümberkirjutamisest ja lõpetades teenuste valideerimisega testimiskeskkonnas. Lõppfaasis hindame lahenduse kvaliteeti tehniliste testide ning arendajatelt ja kasutajatelt kogutud tagasiside põhjal.

3 Analüüs

See peatükk hõlmab lahenduse väljatöötamiseks vajalikke teoreetilisi aluseid ja tehnoloogilist analüüsi. Peatükk algab vaadeldavuse kontseptsiooni ja konteinerorkestreerimise põhimõtete tutvustamisega, millele järgneb turul saadaolevate alternatiivsete seire- ja orkestreerimissüsteemide ülevaade. Lisaks esitatakse valitud tehnoloogiate detailne analüüs ning argumentatsioon tehtud valikute põhjendamiseks.

3.1 Vaadeldavuse kontseptsioon ja komponendid

Probleemide nagu standardiseerimata logid ja puudulik ülevaade süsteemi üldisest seisundist lahendamiseks ei piisa olemasoleva seiresüsteemi lihtsast täiustamisest. Kaasaegsete hajusarhitektuuride, nagu mikroteenused, keerukus nõuab uut lähenemist ehk vaadeldavust.

Käesolev peatükk loob teoreetilise aluse, mis on vajalik skaleeruva seire- ja analüüsilahenduse disainimiseks. See peatükk selgitab vaadeldavuse kontseptsiooni, toob esile selle erinevused traditsioonilisest jälgimisest ja kirjeldab kolme peamist komponenti, mida peetakse vaadeldavuse aluseks. Nende kontseptsioonide mõistmine on kriitilise tähtsusega, et hinnata, miks valitud tehnoloogiad ja meetodid on Kooud ja Arete süsteemide jaoks kõige sobivamad, ja samuti tagada, et loodava lahenduse abil saaksid arendajad ja administraatorid süsteemi käitumisest põhjaliku ülevaate.

3.1.1 Sissejuhatus vaadeldavusse

Tarkvarasüsteemide kontekstis tähendab vaadeldavus võime mõista keeruka süsteemi sisemist olekut selle välise väljundi põhjal. See hõlmab näiteks logisid, mõõdikuid ja jälgi, mis võimaldavad teha järeldusi süsteemi käitumise kohta ilma selle sisemisi komponente otseselt uurimata [3].

Kuigi mõisted seire ja vaadeldavus on omavahel tihedalt seotud ja neid kasutatakse sageli sünonüümidena, on nende tähenduses oluline erinevus.

Seire on traditsiooniline lähenemine, mille eesmärk on jälgida eelnevalt defineeritud mõõdikuid ja koguda andmeid, et vastata teadaolevatele küsimustele. Seire tugineb hoiatusteadetele, mis annavad märku teadaolevatest riketest või olukordadest, mida on osatud ette näha ja mille jaoks on seirereeglid loodud [4].

Vaadeldavus läheb sammu võrra edasi. See on suunatud tundmatute probleemide uurimisele ja aitab vastata küsimustele, mida ei osatud süsteemi loomisel ette näha. Kui seire ütleb, mis on katki, siis vaadeldavus aitab mõista, miks see katki on. Eriti oluliseks muutub see hajussüsteemides, kus üks kasutaja päring võib läbida mitmeid iseseisvaid mikroteenuseid. Sellises keskkonnas ei ole rikke põhjus tihti ilmne ning probleemi algallika leidmine nõuab võimet analüüsida süsteemi käitumist tervikuna [4].

Kokkuvõttes ei ole vaadeldavus tööriistade kogum, vaid süsteemi arhitektuuri ja disaini lahutamatu osa. See eeldab, et süsteem on ehitatud viisil, mis toodab kvaliteetseid andmeid, mis võimaldavad arendajatel ja süsteemiadministraatoritel oma tööd tõhusalt teha, kiirendades vigade tuvastamist ja parandamist ning pakkudes sügavamat arusaamist süsteemi toimimisest.

3.1.2 Vaadeldavuse komponendid

Nagu varem selgitati, võimaldab vaadeldavus süsteemist sügavamalt aru saada ja ootamatute probleemide põhjuseid mõista. Selle saavutamiseks kasutavad tarkvarasüsteemid kolme peamist tüüpi andmeid: mõõdikuid, logisid ja jälgi.

- **Mõõdikud** (ingl. k *metrics*) on süsteemi kohta regulaarselt kogutavad numbrilised andmed, mis kirjeldavad süsteemi mingi aspekti praegust olekut või muutust. See on kokkuvõtlik teave, mis annab ülevaate süsteemi seisundist ja jõudlusest. Mõõdikud on näiteks protsessori koormus, mälukasutus, süsteemi poolt sekundis töödeldud päringute arv või keskmine vastuseaeg päringu kohta. Mõõdikud sobivad kõige paremini süsteemiressursside jälgimiseks ja teadete määramiseks teatud lävede ületamise korral [3]. Üldiselt aitavad mõõdikud kiiresti tuvastada suurenenud koormust või ebatavalist käitumist.
- **Logid** (ingl. k *logs*) on süsteemi genereeritud sündmuste kirjed, mis dokumenteerivad konkreetseid toiminguid või süsteemi oleku muutusi. Erinevalt mõõdikutest, mis esindavad kokkuvõtlikke numbrilisi andmeid, logid sisaldavad tekstiteavet. Need

pakuvad konteksti ja aitavad täpselt kindlaks teha, mis konkreetsel ajahetkel juhtus [3]. Et tagada logisõnumite ühtsus ja hõlbustada nende töötlemist erinevate süsteemide poolt, on välja töötatud spetsiifilised protokollid. Näiteks *Syslog*-i protokoll, mida kirjeldab RFC (*Request for Comments*) 5424 [5], määratleb standardiseeritud viisi logisõnumite loomiseks ja edastamiseks arvutisüsteemides. See standard hõlmab olulisi andmeid, nagu sündmuse prioriteet, ajatempel, sõnumi genereerinud seade või rakendus ning võimaluse lisada struktureeritud andmeid täpsema ja masinloetavama konteksti jaoks. Hea logi sisaldab lisaks sündmuse kirjeldusele ka asjakohast abiteavet, näiteks kasutaja või päringu identifikaatorit või veakoodi [5]. Kooud ja Arete süsteemide puhul on logid üliolulised näiteks õpilase koodi esitamisega seotud vigade täpse algpõhjuse määramiseks.

- **Jäljed** (ingl. k *traces*) ehk hajutatud jälgimine on mehhanism, mis võimaldab jälgida ühe päringu või toimingu kogu teed süsteemi erinevate teenuste kaudu. Selle tee iga etapp salvestatakse eraldi segmendina, mis sisaldab teavet algus- ja lõppaja, kestuse ning seotud teenuse kohta [3]. Jälgimine on hindamatu väärtusega keerukates hajussüsteemides, kus üks kasutaja tegevus (näiteks koodi esitamine ja testimine) läbib mitut mikroteenust. See aitab visualiseerida teenuste vahelist interaktsiooni, tuvastada kitsaskohti ja mõista, kus täpselt oli päring ressursimahukas või tekkis viga.

Nende kolme andmetüübi kasutamine loob kindla aluse süsteemi usaldusväärsusele ja läbipaistvusele. Mõõdikud pakuvad kiiret ülevaadet, logid pakuvad üksikasjalikku konteksti ja jäljed koondavad tervikpildi, võimaldades keeruliste süsteemide tõhusat analüüsi ja haldamist.

3.1.3 Vaadeldavuse komponentide koosmõju

Kuigi mõõdikud, logid ja jäljed pakuvad igaüks väärtuslikku teavet süsteemi kohta, on nende tõeline potentsiaal realiseeritav vaid siis, kui neid kasutatakse koos. Nad täiendavad teineteist ning moodustavad tervikliku andmekogu, mis võimaldab süsteemi käitumist täpselt analüüsida ja probleeme süvitsi diagnoosida.

Kujutame ette tüüpilist stsenaariumi Kooud ja Arete süsteemides, kui tekib probleem:

1. **Mõõdikud annavad esialgse hoiatuse**. Süsteem näitab, et testijärjekorra ooteaeg

on ootamatult pikenenud, samal ajal kui Arete testimisteenuse veamäära mõõdik on samuti märkimisväärselt suurenenud. See annab kiire ülevaate süsteemi üldisest seisukorrast, kuid ei näita veel täpset põhjust.

2. **Jäljed aitavad probleemi lokaliseerida**. Kasutades hajutatud jälgimise lahendust, saavad arendajad uurida viimase aja aeglaseid või ebaõnnestunud koodiesitusi. Nad tuvastavad konkreetse jälje, mis näitab, et päring kulutab ootamatult palju aega mingis teenuses ja et just see teenus tagastab vigu. Jälg aitab visualiseerida kogu päringu teekonda läbi teenuste ja näitab koheselt, milline komponent on kitsaskohaks.
3. **Logid annavad detailse vea põhjuse**. Kui süsteemi arendaja on tuvastanud probleemse teenuse ja konkreetsed päringud jälgede abil, saab ta minna selle teenuse logidesse. Kasutades jälgereast saadud päringu identifikaatorit, saab ta logidest filtreerida ja leida täpsed veateated, mis ilmnesid teenuses. Need logikirjed annavad selgituse vea põhjuse kohta, pakkudes vajalikku konteksti lahenduse leidmiseks.

Vaadeldavuste põhimõtete rakendamine ja kolme põhikomponendi integreerimine võib parandada süsteemi stabiilsust, kiirendada arendusprotsessi ja lihtsustada süsteemihaldust.

3.2 Konteinerorkestratsiooni süsteem

Eelmises peatükis kirjeldatud arhitektuursete probleemide lahendamiseks vajab süsteem konteinerorkestratsiooni süsteemi, mis toetab koormuse jaotamist mitme serveri vahel ja automaatset taastumist. Tallinna Tehnikaülikoolis on toimunud sisemised arutelud klastripõhisele infrastruktuurile ülemineku kohta, mistõttu on Kubernetes kui valdkonna enim kasutatav konteinerorkestratsiooni süsteem loogiline ja põhjendatud valik. Käesolev peatükk selgitab, miks praegune ühe serveri lahendus neile nõuetele ei vasta, ning tutvustab Kubernetese kui kaasaegse konteinerorkestreerimissüsteemi põhimõtteid ja komponente.

3.2.1 Konteinerorkestratsiooni kontseptsioon ja Kubernetes

Kooud ja Arete süsteemid töötavad praegu Dockeri konteineritena ühel serveril. Selline arhitektuuriline lahendus sobib hästi väikesemahuliseks tootmiseks, kuid süsteemi kasvades ilmnevad olulised piirangud.

Esimene ja kõige kriitilisem probleem on ühe tõrkepunkti olemasolu. Kuna kõik teenused

töötavad ühel serveril, põhjustab selle serveri rike kogu süsteemi kättesaamatuse. Kooud ja Arete kontekstis tähendab see, et eksamite ajal võib teenuse katkestus mõjutada kõiki üliõpilasi korraga, millel on otsene negatiivne mõju õppetööle. See on infrastruktuuriline piirang, mis tuleneb süsteemi praegusest ühe serveri lahendusest.

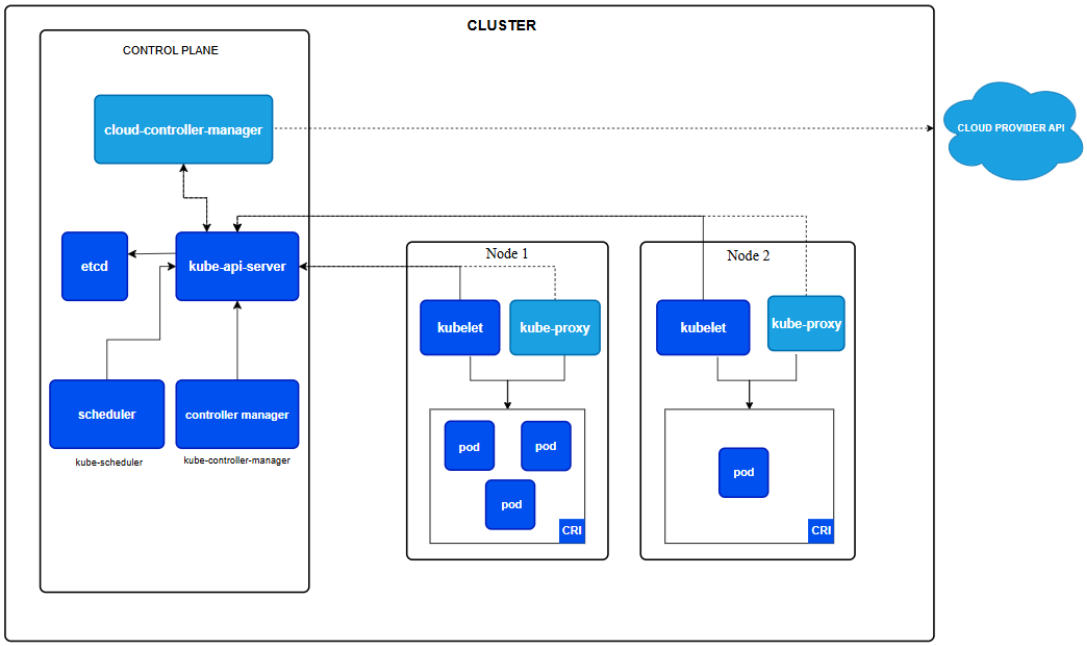
Teine probleem on skaleeritavuse puudumine. Praegune ühe serveri arhitektuur ei võimalda koormuse kasvades automaatselt lisada uusi teenuse eksemplare. Koormuse tipphetkedel, näiteks eksamite ajal, võib süsteem muutuda aeglaseks ning teenused ei suuda piisavalt kiiresti päringutele vastata.

Kolmas kitsaskoht on käsitsi halduse keerukus: teenuste uuendamine, taaskäivitamine ja seisundi jälgimine nõuavad arendajatelt otsest sekkumist serveris. Puudub automaatne taastumine (ingl. k *self-healing*), mis tähendab, et kui mõni teenus katkeb, peab administraator selle käsitsi üles tõstma. Kuigi teenuse käsitsi taaskäivitamine on tehniliselt lihtne toiming, peidab see endas olulise probleemi: administraator on sunnitud reageerima tõrkele selle asemel, et analüüsida selle põhjust. Öise või nädalavahetuse tõrke korral võib süsteem jääda kättesaamatuks kuni administraator probleemi märkab ja reageerib - nagu peatükis 2.1 kirjeldatud, on sellised olukorrad põhjustanud kuni 14-tunniseid katkestusi.

Automaatse taastumise puudumine mõjutab süsteemi haldust mitmel tasandil. Kui teenus taaskäivitatakse käsitsi, kulub administraatori tähelepanu ja aeg teenuse üles tõstmisele, mitte probleemi algpõhjuse leidmisele. Ideaalis peaks süsteem taastuma iseseisvalt, võimaldades administraatoril kohe pärast tõrke tuvastamist asuda analüüsima, miks teenus katkes. Kubernetese *self-healing* mehhanism lahendab just selle probleemi - teenus taaskäivitub automaatselt ning administraator saab kohe pöörduda logide poole, et mõista tõrke algpõhjust, ilma et peaks esmalt muretsema teenuse kättesaadavuse pärast. See seos *self-healing* ja logimise vahel on eriti oluline: uus logimissüsteem võimaldab näha täpselt millal ja miks teenus katkes, ning automaatne taastumine tagab, et süsteem jätkab tööd seni kuni administraator probleemi lahendab.

Kubernetes on avatud lähtekoodiga konteinerorkestratsiooni süsteemile, mis on algselt välja töötatud Google poolt ja mille haldamine on üle antud Cloud Native Computing Foundationile [6] [7]. Kubernetes lahendab kirjeldatud kitsaskohad süsteemselt: see pakub automaatset taastumist, horisontaalset skaleerimist ja deklaratiivset konfiguratsioonihaldust,

mis vähendab vajadust käsitsi sekkumise järele [6] [8]. Kubernetese töö põhineb klastri mudelil, kus infrastruktuur jaguneb juhtimistasandiks (ingl. k *control plane*) ja töösõlmedeks (ingl. k *nodes*). Juhtimistasand vastutab kogu klastri seisundi jälgimise ja soovitud oleku tagamise eest, töösõlmed on aga serverid, millel konteinerid tegelikult käivituvad. Kogu infrastruktuur kirjeldatakse deklaratiivselt YAML-failidena, milles määratletakse soovitud olek - kui mõni komponent lakkab töötamast, taaskäivitab Kubernetes selle automaatselt ilma inimese sekkumiseta. Kubernetese klastri üldine ülesehitus on kujutatud joonisel 3.



Joonis 3. Kubernetese klastri üldine ülesehitus [9].

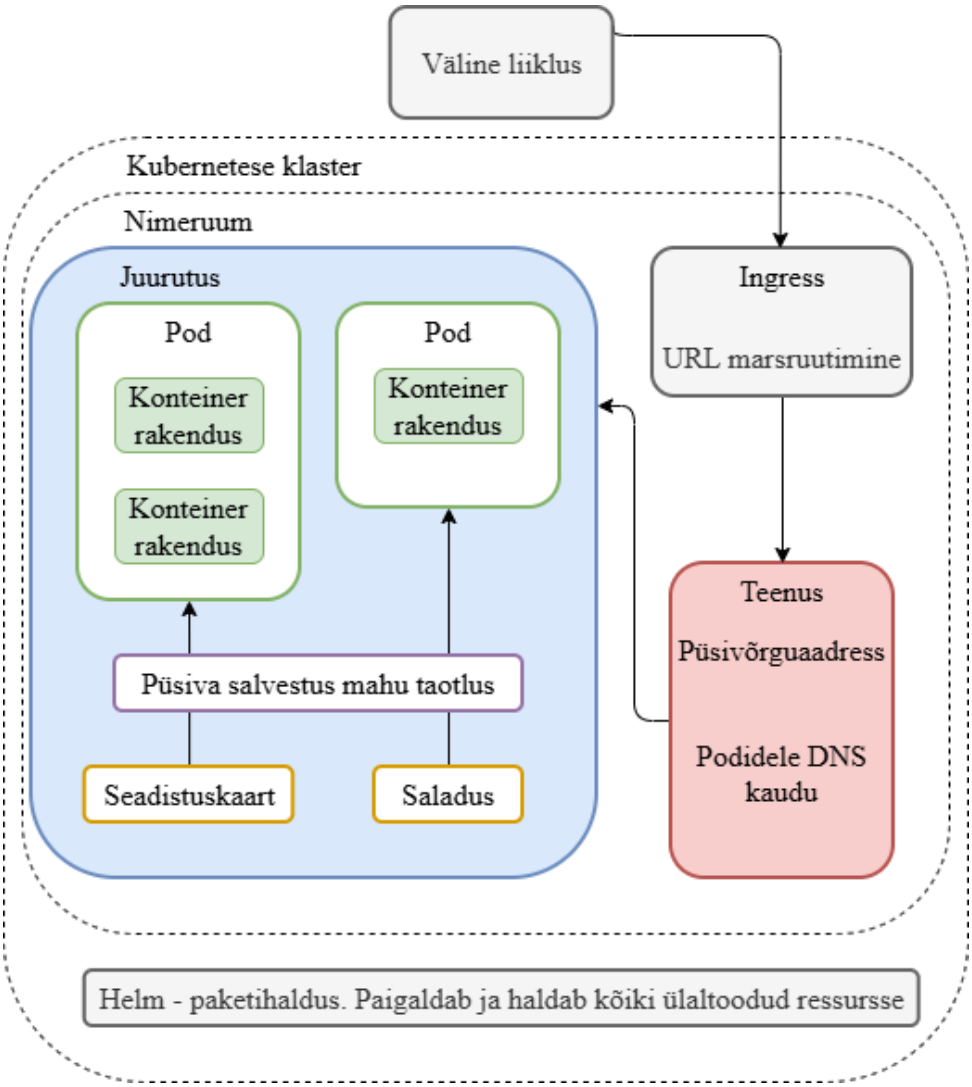
Kubernetese ressursside paremaks mõistmiseks kirjeldatakse järgnevalt põhikomponente, mida Kooud ja Arete süsteemide migratsioonil kasutatakse.

- **Nimeruum** (ingl. k *namespace*) on Kubernetese ressursside loogiline eraldusühik, mis võimaldab grupeerida seotud komponente ja rakendada neile ühiseid piiranguid [10].
- **Pod** on Kubernetese väikseim juurutatav üksus, mis sisaldab ühte või mitut konteinerit. Kõik konteinerid ühes podis jagavad sama võrguaadressi ja salvestusruumi - praktiliselt vastab üks pod ühele teenuse eksemplarile [11].
- **Juurutus** (ingl. k *deployment*) on ressurss, mis haldab podide elutsüklit deklaratiivselt. Juurutus määrab, mitu replikat peab töötama, millist konteineripilti kasutada ja kuidas

uuendused läbi viia. Lisaks sisaldab juurutus elavuse- ja valmisolekuproove (ingl. k *liveness ja readiness probe*), mis võimaldavad Kubernetesel automaatselt tuvastada, kas teenus töötab korrektselt, ja vajadusel pod taaskäivitada või liiklus ümber suunata [12].

- **Teenus** (ingl. k *service*) on abstraktsioon, mis annab podide grupile püsiva võrguaadressi. Kuna podide IP-aadressid muutuvad iga taaskäivituse korral, on teenuse kasutamine hädavajalik teenuste omavaheliseks suhtlemiseks. Teenused leiavad üksteist klastri siseselt Kubernetese DNS-i kaudu[13].
- **Seadistuskaart** (ingl. k *ConfigMap*) ja saladus (ingl. k *secret*) on ressursid konfiguratsiooniväärtuste ja tundlike andmete hoidmiseks väljaspool konteineripilti. Seadistuskaart sisaldab avalikke seadeid nagu andmebaasi aadress või RabbitMQ hostinimi. Saladus hoiab tundlikke andmeid nagu paroolid, API võtmed ja SSH-võtmed, eraldades need konteineripiltidest ja lähtekoodihoidlast. Mõlemad ressursid süstitakse konteineritesse keskkonnamuutujatena, mis tagab, et tundlikud andmed ei ole konteineripiltidesse sisse kodeeritud [14] [15].
- **Püsiva salvestusmahu taotlus** (ingl. k *PersistentVolumeClaim*) on ressurss püsiva salvestusruumi haldamiseks. Erinevalt konteinerite ajutisest failisüsteemist säilivad püsiva salvestusmahu taotlusel olevad andmed podi taaskäivituste ja uuenduste üle [16].
- **Ingress** on ressurss, mis haldab välist HTTP-liiklust klastri teenustele. Nginx Ingress võimaldab määratleda URLide põhiseid marsruutimisreegleid, et suunata erinevad päringud õigetele teenustele [17].
- **Helm** on Kubernetese paketihaldur, mis lihtsustab keerukate rakenduste paigaldamist ja haldamist taaskasutatavate pakettide (*chart*'ide) kaudu [18].

Joonis 4 illustreerib Kubernetese põhikomponentide hierarhiat ja nende seoseid.



Joonis 4. Kubernetese klastri põhikomponendid ja nende seosed.

Andmebaasihalduseks on võimalik kasutada Kubernetese operaatoreid, mis laiendavad platvormi funktsionaalsust spetsiifiliste töökoormuste haldamiseks, näiteks andmebaaside automaatse tõrkeülekandumise ja replikatsiooni jaoks [19].

3.3 Seiresüsteemi turu analüüs

Turul on palju monitooringuteenuseid nagu Splunk, ELK Stack, Grafana, OpenSearch, Prometheus ja teised [20] [21] [22]. Meie eesmärk on luua süsteem, mis kogub ja salvestab süsteemist logisid ja mõõdikuid ning visualiseerib neid paindlikes töölaudades. Valitud teenus peaks olema avatud lähtekoodiga, sest see on tasuta, tulevikus muudetav ning see on kliendi nõue.

3.3.1 ELK

Kõige populaarsem avatud lähtekoodiga monitooringutööriist on ElasticSearchi põhine stack. ELK on kolme avatud lähtekoodiga teenuse kogum: ElasticSearch, Logstash ja Kibana.

1. **ElasticSearch**: ElasticSearch on andmebaas, millel on täisteksti otsingu- ja analüüsimootor; kõik selles andmebaasis olev tekst on tänu indekseerimisele kiiresti ja lihtsalt otsitav.
2. **Logstash**: Logstash on logide agregeerimise tööriist, mis kogub andmeid erinevatest allikatest, teeb transformatsioone ning saadab need edasi erinevatesse väljunditesse. Oluline on märkida, et seda teenust saab hõlpsasti asendada kergema lahendusega Fluentd [23].
3. **Kibana**: Kibana on selle stack'i pealmine kiht, mis visualiseerib logisid töölaudades. Kasutajad saavad luua ja kohandada töölaudu.

Kõik need teenused on asendatavad ja sõltumatud ning neid saab kasutada ka teistes tehnoloogiakomplektides. 2021. aasta alguses teatas Elastic, et alates versioonist 7.11 ei ole ELK enam täielikult avatud lähtekoodiga, kuna mindi üle Apache 2.0 litsentsilt SSPL ja ELv2 litsentsidele. Need litsentsid piiravad levitamist. Õnneks on monitooring mõeldud ainult TalTechi sisekasutuseks, seega kuigi uus litsents ei ole avatud lähtekoodiga, see ei mõjuta meie süsteemi haldamist [24] [20].

Eelised:

- Suur ökosüsteem ja kogukonna tugi
- Täielikult indekseeritud logid võimaldavad kiiret otsingut ja filtreerimist
- Rikkalikud visualiseerimisvõimalused Kibana abil
- Osaliselt avatud lähtekood

Puudused:

- Logide indekseerimine suurendab kettaruumi kasutust
- Ei ole mõeldud mõõdikute kogumiseks ja salvestamiseks
- On olnud juhtum, kus arendaja muutis litsentsi, seega võib see korduda ja teenus

võib täielikult kaotada avatud lähtekoodi staatuse

3.3.2 OpenSearch

Vahetult pärast seda, kui Elastic teatas litsentsimuudatusest, algatas AWS OpenSearch projekti, mis põhineb ELK platvormi avatud lähtekoodiga versioonil. OpenSearch on tasuta ja täielikult avatud lähtekoodiga lahendus, mida haldab AWS, mitte Elastic. Benchmark-uuringute kohaselt on selle jõudlus võrreldav ELK platvormiga ning teatud kategooriates ületab teda [25]. Funktsionaalsuse poolest pakub OpenSearch suurel määral samu eeliseid nagu ELK, kuid sellega kaasnevad ka mitmed sarnased piirangud [22].

Eelised:

- Täielikult indekseeritud logid
- Rikkalikud visualiseerimisvõimalused OpenSearch Dashboards abil
- Täielikult avatud lähtekood

Puudused:

- OpenSearch'il on noor ökosüsteem
- OpenSearch'il on vähem integratsioone ja pistikprogramme, see võib raskendada selle süsteemi integreerimist

3.3.3 Grafana Stack

Grafana on terviklik lahendus, mis koosneb mitmest tihedalt integreeritud avatud lähtekoodiga komponendist. Selle tehnoloogiapaketi juurutamine võimaldab kasutada süstemaatilist jälgimismeetodit, mis hõlmab mõõdikuid, logisid ja jälgi ühes infrastruktuuris [21].

Grafana Stack sisaldab selliseid komponente nagu Grafana visualiseerimisteenus, Loki teenus logide kogumiseks ja salvestamiseks ning Mimir mõõdikute salvestamiseks ja haldamiseks. Sellel on palju erinevaid komponente erinevateks eesmärkideks ning need kõik toimivad ühtses Grafana ökosüsteemis.

Eelised:

- Loki logid võtavad kettal vähem ruumi

- Rikkalikud visualiseerimisvõimalused Grafana abil (üks populaarsemaid lahendusi)
- Täielikult avatud lähtekood
- Grafana ökosüsteem sisaldab kõiki vajalikke tööriistu

Puudused:

- Loki logid on otsitavad ainult metaandmete alusel

3.4 Valitud seiresüsteem

Pärast uurimist ja arutelu kliendiga valisime seireteenuseks Grafana Stacki. Grafana Stacki eelis seisneb eelkõige selle paindlikkuses ja sõltumatusest. Kuna tegemist on avatud lähtekoodiga tarkvaraga, on võimalik kogu lahendus paigaldada oma serverites, mis annab arendajatele kontrolli nii andmete ja turvalisuse kui ka süsteemi konfiguratsiooni üle.

Lisaks tehnilisele sõltumatusele pakub Grafana Stack ka hästi arenenud ökosüsteemi ja põhjalikku dokumentatsiooni, mis lihtsustab nii juurutamist kui ka haldust. Võrreldes alternatiividega, võimaldab Grafana Stack katta kõik kolm vaadeldavuse sammast ühe platvormi piires. See vähendab integreerimiskulusid ja tagab parema kokkusobivust erinevate andmetüüpide vahel.

3.4.1 Grafana

Grafana on keskne kasutajaliides andmete visualiseerimiseks ja analüüsimiseks, mis pakub arendajatele ja haldajatele ülevaate oma süsteemide olekust. See on laialdaselt kasutatav platvorm, mis integreerib erinevad andmeallikad ühte kasutajaliidesesse ja pakub nende põhjal selgeid ja kohandatavaid visualiseeringuid [26].

Kooud ja Arete süsteemide kontekstis lahendab Grafana mitmeid olemasoleva töölaua probleeme. Praegused lahendused piiravad analüüsi ühe päeva andmetega ja pakuvad enamasti staatilisi graafikuid, mis ei toeta sügavamat uurimist. Grafana seevastu võimaldab dünaamilist andmetega manipuleerimist, andes kasutajatele vabaduse valida ajaperioode ja süveneda üksikute sündmustele.

Oluliseks eeliseks on interaktiivsus, mis muudab süsteemi jälgimise passiivsest tegevusest aktiivseks analüüsiprotsessiks. Arendaja saab esitada küsimusi ja neid kontrollida. Näiteks

on võimalik siduda mõõdikud ja logid samasse vaatesse, mis aitab kiiresti tuvastada seoseid süsteemi käitumise ja vigade vahel. Selline korrelatsioon vähendab aega, mis kulub probleemi algpõhjuse leidmiseks.

Lisaks toetab Grafana paindlikku filtreerimist ja andmete grupeerimist, mis võimaldab analüüsida süsteemi erinevatest vaadest. Märgiste (nt teenuse nimi või logitase) kasutamine muudab suurte andmemahtude käsitlemise oluliselt efektiivsemaks. See on eriti oluline hajussüsteemide puhul, kus probleemid võivad avalduda mitme teenuse suhtlemises.

Kokkuvõttes ei ole Grafana ainult visualiseerimise rakendus, vaid oluline analüütiline töörist.

3.4.2 Loki

Logide kogumiseks ja salvestamiseks valisime Grafana Loki, mis on loodud peamiselt suurte logivoogude tõhusaks töötlemiseks. Lahenduse peamine eelis seisneb lähenemisviisis, mis indekseerib ainult logidega seotud märgiseid, mitte logisõnumite teksti. See võimaldab päringuid kiiresti filtreerida ning vähendada oluliselt süsteemi poolt kasutatavat salvestusmahtu [27].

Selline lähenemine teeb Loki eriti sobivaks hajussüsteemide jaoks, kus logide hulk võib kasvada väga suureks ning traditsioonilised täisteksti indekseerimisel põhinevad lahendused muutuvad ressursimahukaks. Kuna indeksit hoitakse minimaalsena, suunatakse suurem osa andmetest kompaktsesse salvestusse, mis omakorda parandab skaleeritavust.

Loki kasutab logide struktureerimiseks voopõhist mudelit, kus iga logivoog on defineeritud kindlate märgiste komplektiga. See tähendab, et kõik sama märgistega logid grupeeritakse ühte voogu, mis võimaldab neid hiljem kiiresti filtreerida ja analüüsida. Märgiste süsteem on paindlik ning võimaldab kirjeldada logide konteksti mitmel tasandil, näiteks teenuse või logitaseme alusel.

Lisaks toetab Loki logide ajalist järjestust, mis võimaldab analüüsida sündmusi nende tekkimise järjekorras. See on oluline eelkõige probleemide diagnoosimisel, kus sündmuste ajaline seos aitab tuvastada vigade algpõhjuseid.

Kokkuvõttes pakub Loki lihtsat, kuid skaleeritavat logide kogumise lahendust, mille eelised

on tõhusus, väike ressursikulu ja kiire päringute töötlemine isegi suurte andmemahtude korral.

3.4.3 Mimir

Mõõdikute töötlemiseks valisime Grafana Mimir. Grafana Mimir on täielikult ühilduv Prometheus-ega, mis tähendab, et see toetab sama mitmemõõtmelist andmemudelit, kus aegridad identifitseeritakse mõõdiku nimede ja võtme-väärtus siltide paaride abil, samuti PromQL päringukeelt [28].

Prometheus on laialdaselt kasutatav tööriist mõõdikute kogumiseks ja monitoorimiseks. Siiski ei toeta see vaikimisi klasterdamist ega horisontaalset skaleerimist. Kõrge käideldavusega seadistustes tuleb mitut Prometheuse instantsi juurutada ja hallata käsitsi, mis suurendab süsteemi keerukust ega paku tõeliselt hajutatud andmesalvestust [29].

Nende piirangute lahendamiseks arendas Grafana Labs välja Grafana Mimiri. Mimir on horisontaalselt skaleeritav, hajutatud salvestussüsteem Prometheuse mõõdikute jaoks. See pakub sisseehitatud replikatsiooni, *sharding*'ut ja kõrget käideldavust, võimaldades usaldusväärset pikaajalist suurte mõõdikuandmete mahtude salvestamist [28].

Selles arhitektuuris vastutab Prometheus endiselt mõõdikute kogumise eest, samas kui Grafana Mimir tegeleb skaleeritava salvestuse ja päringutega, parandades monitoorimissüsteemi töökindlust ja veataluvust.

3.5 Kubernetese alternatiivid

Käesolevas peatükis analüüsitakse Kubernetese peamisi alternatiive, et põhjendada lõplikku valikut. Kuna Tallinna Tehnikaülikoolis on toimunud sisemised arutelud klastripõhisele infrastruktuurile ülemineku kohta ning Kubernetes on selles kontekstis kõige loogilisem valik, analüüsitakse siinkohal alternatiive eelkõige selleks, et selgitada, miks teised lahendused ei sobi ning kinnitada Kubernetese valiku põhjendatust.

Docker Swarm on Dockeri sisseehitatud režiim (ingl. k *mode*) [30], mis võimaldab käivitada lihtsa konteineriklastri. Selle peamine eelis on madal õppimiskõver, kuna see kasutab sama tööriistu ja konfiguratsioonisüntaksit mis Docker. Siiski on Docker Swarmil olulised piirangud: see ei toeta keerukaid juurutusstrateegiaid, sellel puudub sisseehitatud

autoskaleerimine ning selle ökosüsteem on võrreldes Kubernetesega oluliselt väiksem [30].

HashiCorp Terraform on infrastruktuuri haldamise tööriist, mis on laialdaselt tuntud oma lihtsuse poolest. Terraform võimaldab simuleerida mõningaid klastri funktsioone, kuid ei ole täisväärtusliku konteinerorkestratsiooni platvorm - sellel puuduvad Kubernetesele omased võimalused nagu automaatne taastumine, sisseehitatud koormuse tasakaalustamine ja konteinerite elutsükli haldus. Terraform sobib pigem infrastruktuuri ettevalmistamiseks kui konteinerite orkestreerimiseks [31].

HashiCorp Nomad [32] on üldotstarbeline töökoormuse orkestraator, mis toetab lisaks konteineritele ka teisi töökoormuse tüüpe nagu virtuaalmasinad ja eraldiseisvad rakendused. Nomad on tuntud oma lihtsuse ja paindlikkuse poolest, kuid selle ökosüsteem on oluliselt väiksem kui Kubernetese oma. Nomad ei paku sisseehitatud lahendusi selliste komponentide jaoks nagu Ingress või püsiva salvestusmahu taotlus, mis tähendab et need tuleb eraldi tööriistadega katta. Samuti puudub Nomadil laialdane kogukonna tugi ja kolmandatest osapooltest operaatorite ökosüsteem.

Kubernetes [6] on konteinerorkestratsiooni valdkonna üldtunnustatud standard [7], mille arendas algselt Google ning mille haldamine anti seejärel üle Cloud Native Computing Foundationile [7]. Kubernetes pakub täielikku konteinerorkestratsiooni lahendust koos automaatse taastumise, horisontaalse skaleerimise, deklaratiivse konfiguratsioonihalduse ja ulatusliku ökosüsteemiga. Võrreldes eelnevalt kirjeldatud alternatiividega on Kubernetes ainus süsteem, mis katab kõik käesoleva töö nõuded.

Võrreldes kõigi eelnimetatud alternatiividega on Kubernetes ainus süsteem, mis katab kõik käesoleva töö nõuded - automaatne taastumine, horisontaalne skaleerimine, deklaratiivne konfiguratsioonihaldus ja ulatuslik ökosüsteem. Lisaks on Kubernetes kooskõlas Tallinna Tehnikaülikooli sisemiste aruteludega klastripõhisele infrastruktuurile ülemineku kohta, mis muudab selle valiku veelgi põhjendatumaks.

3.6 Kubernetese valimine

Eelmises peatükis kirjeldatud alternatiivide analüüs näitas, et Docker Swarm, Terraform ja Nomad ei kata käesoleva töö nõudeid - neil puuduvad vajalikud võimalused nagu automaatne taastumine, sisseehitatud Ingress ja ulatuslik kolmandatest osapooltest operaatorite

ökosüsteem. Kubernetes osutus ainsaks platvormiks, mis vastab kõigile nõuetele.

Kubernetes on konteinerorkestratsiooni valdkonna üldtunnustatud standard, millel on suurim kogukond ja ökosüsteem [6] [7]. See tähendab, et enamik levinumatest infrastruktuuri probleemidest on juba dokumenteeritud ja lahendatud. CloudNativePG operaator PostgreSQL klastrite haldamiseks on Kubernetese jaoks spetsiaalselt loodud laiendus, mis pakub automaatset tõrkeülekandumist ja replikatsiooni - funktsionaalsust, mida teiste süsteemide puhul tuleks eraldi tööriistadega katta.

Kuna kogu meie arendustöövoog on üles ehitatud GitLabi ümber - alates koodihaldusest kuni konteineripiltide registrini - on GitLabi natiivne integratsioon Kubernetesega GitLabi Kubernetese agendi kaudu [33] oluline mugavusargument. See võimaldab luua tervikliku pideva lõimimise ja tarnimise torustiku (ingl. k *CI/CD pipeline*), kus koodimuudatused käivitavad automaatselt uue versiooni juurutuse klastris, ilma et oleks vaja eraldi tööriiste või lisaseadistusi hallata.

Oluline tegur Kubernetese valikul oli ka see, et organisatsiooni sees käib Kubernetese klastri hindamine, mis toetab käesolevas töös tehtud valikut kui kõige sobivamat platvormi. Lisaks planeeritakse ülikooli siseselt Kubernetese klastri seadistamist, mis tähendab et käesolevas töös loodud konfiguratsioonid on tulevikus vahetult rakendatavad ning ei nõua platvormi vahetust.

Kokkuvõttes lahendab Kubernetes kõik kolm praeguse infrastruktuuri põhiprobleemi. Automaatne taastumine elimineerib ühe tõrkepunkti riski, horisontaalne skaleerimine võimaldab koormuse tipphetkedel lisada uusi teenuse eksemplare ning deklaratiivne konfiguratsioonihaldus koos pideva lõimimise ja tarnimisega vähendab käsitsi halduse vajadust.

3.7 Objektisalvestus

Tudengite koodifailide ja testimistulemuste püsivaks salvestamiseks kasutatakse hajutatud objektisalvestuse lahendust. Objektisalvestus on andmesalvestuse kontseptsioon, kus failid salvestatakse tasaste objektidena koos metaandmetega, erinevalt traditsioonilisest failisüsteemist. Amazon S3 (ingl. k *Simple Storage Service*) on Amazoni poolt loodud objektisalvestuse teenus [34], millest on saanud de facto standard objektisalvestuse vald-

konnas. S3 protokoll määratleb standardse liidese objektisalvestusega suhtlemiseks, mida toetavad paljud avatud lähtekoodiga lahendused.

Objektisalvestuse alternatiivide analüüsimisel vaadeldi kahte peamist avatud lähtekoodiga lahendust: MinIO ja Garage. MinIO on laialt tuntud S3-ühilduv objektisalvestuse lahendus, kuid selle litsents muutus poolpropritaarseks ning tasuta Community Edition on funktsionaalsuselt tugevalt piiratud, mis muudab selle ebasobivaks pikaajalise lahendusena [35]. Garage on täielikult avatud lähtekoodiga hajutatud objektisalvestuse süsteem, mis on ühilduv Amazon S3 liidesega [36]. See tähendab, et kõik rakendused mis toetavad S3 protokolli saavad Garage'iga ilma muudatusteta suhelda. Objektisalvestuse lahendusena valiti Garage, kuid selle integreerimine teenustesse jääb edasise arendustöö osaks.

4 Lahenduse arendus

Pärast teoreetilise analüüsi läbiviimist ja tehnoloogilise lahenduste komplekti valimist määratleti peamised nõuded, mis tuleb projekti käigus täita:

- paigaldada ja seadistada Grafana Stack serverikeskkonnas;
- luua ja integreerida ühtne logimisstandard;
- kavandada andmekogumise arhitektuur ja andmevood;
- rakendada Loki logikogumise lahendus rakenduste koodibaasis;
- seadistada mõõdikute kogumise protsess;
- luua kasutajasõbralik logide ja mõõdikute visualiseerimislahendus;
- kavandada Kubernetese klastri arhitektuur koos nimeruumide, juurutuste ja teenustega;
- migreerida teenuste konfiguratsioonid Podman Quadlet failideks;
- juurutada teenused Kubernetese platvormile koos initsialiseerimisakonteinerite ja seisundikontrollidega;
- seadistada andmebaasihaldus CloudNativePG operaatori abil;
- konfigureerida Nginx Ingress liikluse marsruutimiseks;
- valideerida Kubernetese konfiguratsioonid testimiskeskkonnas.

4.1 Andmekogumise arhitektuur

Selles alapeatükis kirjeldatakse, kuidas süsteemi erinevatest komponentidest pärinevad andmed kogutakse, edastatakse, salvestatakse ning tehakse kättesaadavaks analüüsiks ja visualiseerimiseks.

Valitud tehnoloogiad põhinevad lähenemisel, kus kõik teenused edastavad oma andmed ühte kogumispunkti.

Logide kogumise ja salvestamise eest vastutab Loki. Teenused saadavad oma logisid otse Loki API-sse. Oluline on see, et teenused genereerivad logisid struktureeritud JSON

(*JavaScript Object Notation*) vormingus. See lähenemisviis annab teenustele kontrolli logide sisu üle ja tagab järjepideva logimise kogu süsteemis.

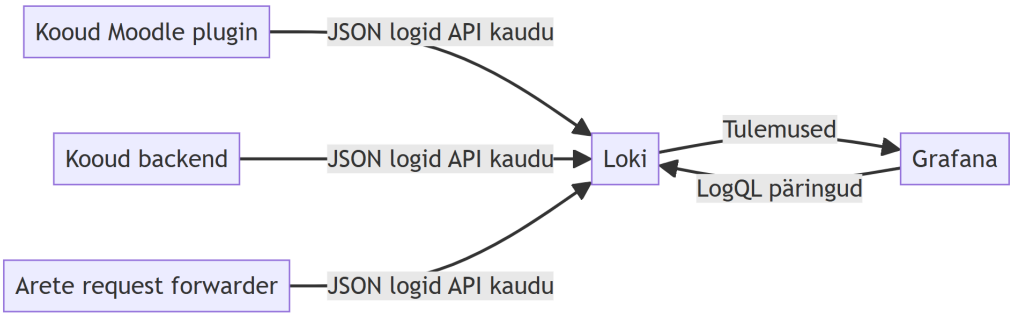
Mõõdikute kogumiseks kasutab süsteem päringutel põhinevat lähenemisviisi. Iga teenus rakendab teeki, mis loob teenuse sees otspunkti, mis väljastab reaalajas andmeid teenuse praeguse oleku kohta ja Prometheus küsib perioodiliselt nende andmete värskendusi.

Lisaks standardsetele süsteemsetele mõõdikutele, nagu protsessori koormus või mälukasutus, võimaldab see lahendus rakendada ka kohandatud mõõdikuid, mis on seotud konkreetse äriloogikaga. Näiteks on võimalik luua loendurid, mis jälgivad koodiesitluste arvu.

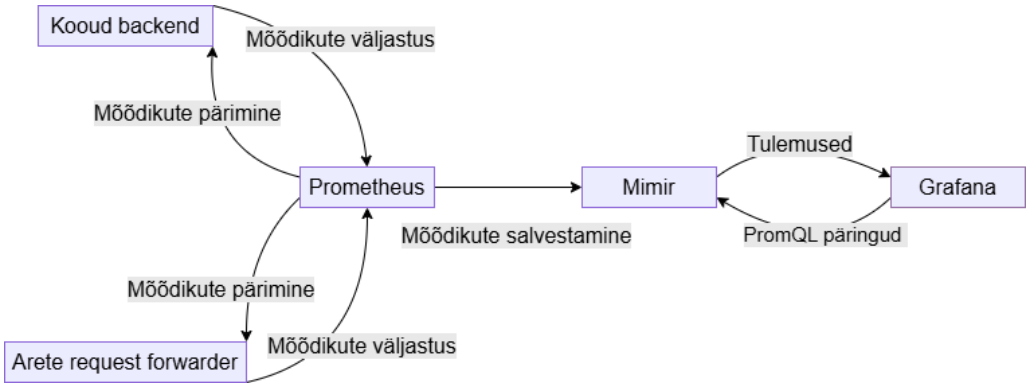
Prometheus kogub need andmed kokku ja edastab need pikaajaliseks säilitamiseks Mimirile, mis on horisontaalselt skaleeritav andmehoidla. Seejärel ühendatakse Mimir Grafanaga kui andmeallikas.

Grafana toimib süsteemi keskse kasutajaliidesena, mis ühendub andmeallikana Loki ja Mimiriga. Arendajad ja administraatorid saavad Grafanas päringukeele abil esitada päringuid salvestatud andmete kohta, luua filtreid ja visualiseerida andmeid paindlike töölaua abil.

Joonis 5 kirjeldab logide ning joonis 6 mõõdikute edastamist teenustest visualiseerimiseks kesksesse asukohta.



Joonis 5. Logide kogumise ja visualiseerimise arhitektuur.



Joonis 6. Mõõdikute kogumise ja visualiseerimise arhitektuur.

4.2 Ühtne logimisstruktuur

Hajutatud süsteemide kontekstis on ühtne logistruktuur ülioluline süsteemi käitumise arusaadavuse ja analüüsi tagamiseks. Varem tuvastatud probleemid, nagu mittestandardsed logid ja raskused teabe leidmisel, tulenevad peamiselt sellest, et erinevad teenused genereerivad logisid erinevates vormingutes. See raskendab logide kogumist, filtreerimist ja analüüsi.

Nende probleemide lahendamiseks valisime kõigi teenuste jaoks ühtse JSON põhise logimisstruktuuri. See võimaldab esitada logisid struktureeritud vormingus, mis on loetav nii arvuti kui ka inimese poolt. See integreerub sujuvalt valitud logide kogumise lahendusega, kuna Loki API nõuab logide edastamist kindlas JSON struktuuris (vt lisa 3). Selline struktuur võimaldab saata ühe päringuga suure hulga logisid, mis on grupeeritud märgiste järgi. Neid gruppe võib omakorda olla mitu. Selline lähenemine vähendab serveri koormust, kuna päringute arv on väiksem.

Iga logikirje koosneb kahest osast: metaandmetest ehk märgistest, mida kasutatakse indekseerimiseks ja kiireks filtreerimiseks, ning logirea sisust, mis sisaldab detailset sündmuse kirjeldust.

Valitud struktuuris on järgmised põhilised märgised:

- *service_name* - määrab, millisest teenusest log pärineb
- *log_level* - kirjeldab logi taset (nt *INFO*, *WARNING*, *ERROR*, *DEBUG*)

Logirea sisus kasutatame ühtset JSON struktuuri, mis sisaldab järgmisi välju:

- *timestamp* – logi loomise aeg
- *message* – inimloetav logi kirjeldus
- *request_id* – unikaalne identifikaator, mis seob kõik kasutaja sama tegevuse logid
- *event_name* – sündmuse nimetus
- *data* – täiendav struktureeritud info (nt kasutaja, kursuse identifikaatorid)

See lähenemisviis lahendab eelnevad probleemid mitmel tasandil. Esiteks tagab ühtne struktuur, et kõik teenused kirjeldavad sündmusi identselt, kõrvaldades mittestandardsete logide probleemi. Teiseks märgiste kasutamine võimaldab Lokis logisid kiiresti teenuse ja logimistaseme järgi filtreerida, ilma et peaks kogu logiteksti töötlema (vt lisa 5).

Tulemuseks on süsteem, kus logid ei ole enam lihtsalt tekstilised kirjed, vaid analüüsitav andmestik, mis toetab efektiivset vigade tuvastamist, süsteemi käitumise mõistmist ning vaadeldavuse põhimõtete rakendamist.

4.3 Grafana Stacki paigaldamine ja seadistamine

Seiresüsteemi juurutamiseks valisime paigaldusviis otse serveri operatsioonisüsteemi, et tagada maksimaalne kontroll ressursside üle. Kogu seirekomplekt, mis koosneb Grafana, Loki, Mimiri ja Prometheuse teenustest, konfigureerisime seadistusfailide abil.

Süsteemi keskne element on Grafana, mis koondab andmeid teistest teenustest. Logide haldamiseks paigaldasime Grafana Loki, konfigureerisime selle keskendudes andmete säilitamisele. Konfigureerisime Loki kasutama kohalikku failisüsteemi, mis salvestab nii logide indekseid kui ka andmekogumeid. Kuna süsteemil on piiratud kettaruum, rakendasime andmete säilitamise reeglit, mis kustutab automaatselt logid, mis on vanemad kui 168 tundi. See tagab, et serveri mälu ei täitu aegunud andmetega.

Mõõdikute pikaajaliseks salvestamiseks kasutasime Grafana Mimiri lahendust, mis pakub paremat skaleeritavust kui Prometheus. Konfigureerisime Mimiri andmeid loogiliselt eraldama. Nagu Loki, kasutab Mimir andmete salvestamiseks serveri kohalikku failisüsteemi. Prometheuse roll on koguda andmeid teenuse lõpp-punktidest, sealhulgas nii süsteemikomponentidest kui ka välistest rakendustest.

Kõik seireteenused paigaldasime arenduskeskkonna serveritesse. Turvalisuse ja mugava

ligipääsu tagamiseks konfigureerisime Nginxi pöördproksi. Kõikidele seireteenustele pääseb ligi krüpteeritud HTTPS (*Hypertext Transfer Protocol Secure*) ühenduse kaudu, kasutades kasutades Let's Encrypti sertifikaate, mille hankimiseks ja uuendamiseks kasutasime Certbot klienti [37] [38].

Nginxi konfigureerimisel pöörasime erilist tähelepanu Mimiri ja Loki API päringutele. Lisasime HTTP (*Hypertext Transfer Protocol*) päistele unikaalne identifikaator, et logisid ja mõõdikuid õigesti klassifitseerida ja eristada neid muudest võimalikest andmevoogudest.

Pärast teenuste käivitamist ja võrguseadistuste lõpetamist lisasime andmeallikad Grafana kasutajaliideses käsitsi. See võimaldas alustada visualiseerimistöölaudade loomist ja süsteemi reaalajas jälgimist.

4.4 Logimise süsteemi seadistamine teenustes

Käesolev alapeatükk keskendub uue logimislahenduse integreerimisele süsteemi teenustesse. Järgnevalt anname ülevaade loodud logimisstruktuuri arhitektuurist, logide lokaalse varundamise mehhanismidest ning lahendustest, mis võimaldavad jälgida ja seostada kasutaja päringuid läbi erinevate mikroteenuste.

4.4.1 Logimissüsteemi struktuur

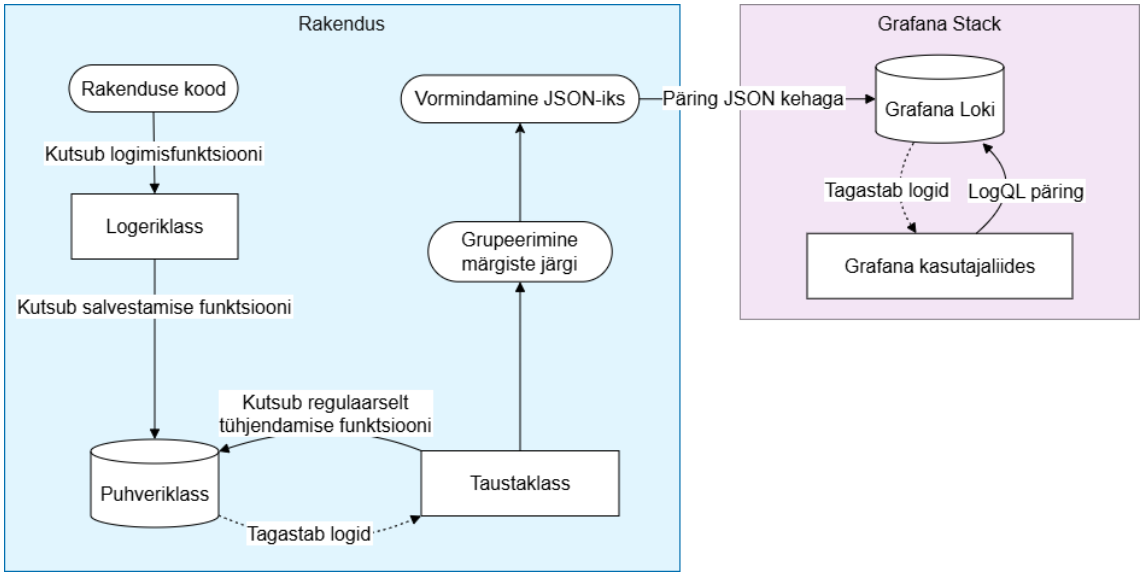
Süsteemi seirevõimaluste tagamiseks arendasime iga teenuse jaoks eraldi logimislahenduse, mis integreerub eelnevalt kirjeldatud keskse arhitektuuriga. Lahenduse aluseks on teenuste sees loodud eraldi logimisklass (*LokiLogger*). Arendajad saavad süsteemis sündmusi logida standardmeetodite abil, näiteks kutsudes funktsioone *info* või *error*. Süsteemi jõudluse säilitamiseks ja võrguliikluse optimeerimiseks ei saadeta iga uut logikirjet kohe seireserverisse, selle asemel salvestatakse sündmused esmalt rakenduse sisesesse puhvrisse. Puhverdamise haldamiseks lõime klass (*LokiLogBuffer*), mille ülesandeks on logide kogumine, hoidmine ning mälu vabastamine andmete edastamiseks.

Selleks, et puhvrisse kogunenud andmed jõuaksid reaalajas logiserverini, rakendasime taustal töötav tsükliline protsess. Kindlaksmääratud ajavahemike tagant käivitub eraldiseisev klass (*LokiBackgroundSender*), mis vastutab puhvri tühjendamise ja andmete vormindamise eest. Töötlemise käigus loetakse puhvrist kõik kogunenud kirjed, puhver tühjendatakse ja

logid rühmitatakse märgiste järgi. Pärast logide rühmitamist koostatakse nendest varasemalt defineeritud struktuurile vastav JSON-objekt.

Lõplikult vormindatud andmepakett edastatakse ühe päringuna otse Loki API-sse. Selline saatmine vähendab oluliselt päringute arvu ja hoiab ära serveri ülekoormamise. Pärast edukat API-päringut on puhver valmis uute sündmuste vastuvõtmiseks. Kui päring ebaõnnestub, salvestatakse logid tagasi puhvrisse ja saatmisprotsessi korratakse aja jooksul veel mitu korda. Loki poolt edukalt vastuvõetud ja indekseeritud andmed on saadaval Grafana keskkonnas, kus arendajad saavad neid oma töölaudadel analüüsida ja visualiseerida (vt lisa 4).

Joonis 7 illustreerib rakenduse logimisprotsessi ja andmete liikumist Grafana Lokisse.



Joonis 7. Logimissüsteemi arhitektuur ja andmevoog.

4.4.2 Logide salvestamine failidesse ja logirotatsioon

Logimissüsteemi töökindluse tagamiseks salvestatakse kõik logid lisaks kesksele logimissüsteemile ka lokaalselt failidesse. See lahendus võimaldab logisid säilitada isegi siis, kui rakendus ootamatu vea tõttu peatub või kui ühendus Loki-ga ajutiselt katkeb.

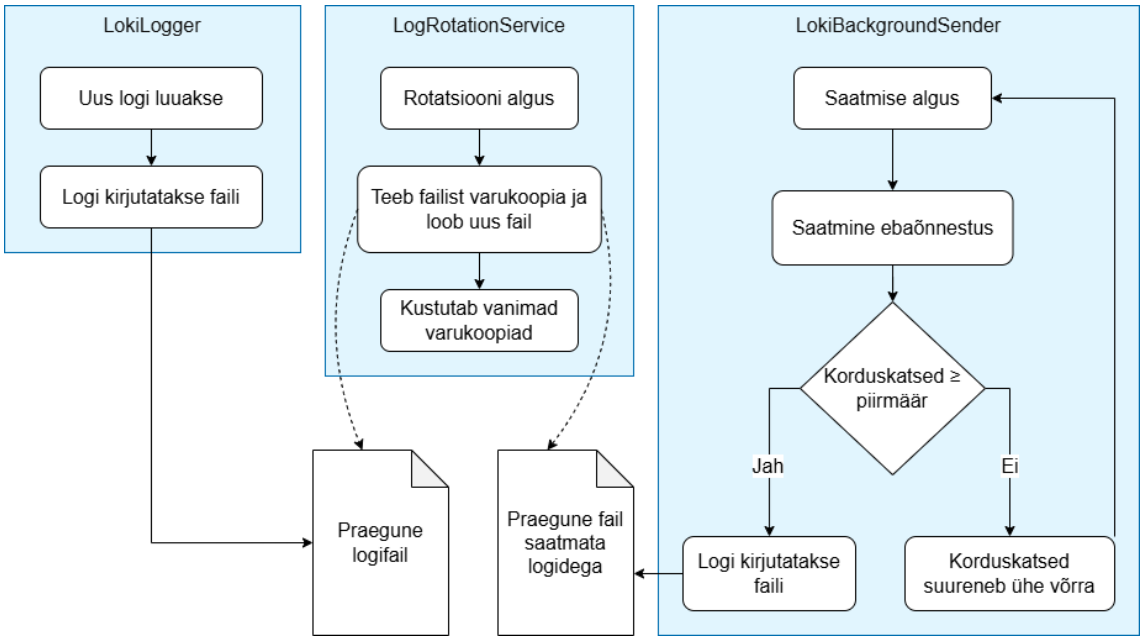
Logide salvestamine toimub kohe nende loomise hetkel. Iga logikirje vormistatakse tekstireana, mis sisaldab ajatemplit, logitaset ning logi sisu JSON-kujul. Seejärel kirjutatakse logi faili. Faili kirjutamine on sünkroniseeritud, et vältida probleeme samaaegse kirjutamise korral mitmest lõimest.

Lisaks põhifailile kasutatakse eraldi faili logide jaoks, mida ei õnnestu isegi pärast mitut katset välisesse süsteemi edastada. See võimaldab hiljem probleemseid logisid eraldi analüüsida ja hoiab ära nende kadumise.

Logifailide lõputu kasvu vältimiseks rakendasime logide rotatsiooni mehhanism. Logide rotatsiooni ajal kontrollitakse perioodiliselt failide vanust. Kui fail ületab määratud vanuse, nimetatakse see ümber varukoopiafailiks, mille nimele lisatakse ajatempel, ja luuakse uus tühi logifail. Lisaks kustutatakse vanad varukoopiad automaatselt, kui nende arv ületab määratud piiri. See aitab kontrollida kettaruumi kasutamist ja tagab, et süsteem ei kogu liigseid logisid.

Selline lähenemine tagab, et logid on säilitatud ka tõrke olukordades ning samal ajal on logifailide haldus automatiseeritud ja ressursisäästlik.

Joonis 8 illustreerib logide salvestamist failidesse ja rotatsiooni.



Joonis 8. Logide salvestamine failidesse ja rotatsioon.

4.4.3 Logide seostamine teenuste vahel

Hajutatud süsteemides läbib üks kasutaja tegevus või päring sageli mitut sõltumatut mikroteenust. Süsteemi üldise käitumise mõistmiseks ja vigade algpõhjuse tuvastamiseks on oluline ühendada erinevate teenuste loodud logid üheks loogiliseks ahelaks.

Selle eesmärgi saavutamiseks lisasime logistruktuuri kaks olulist välja: *event_name* ja *request_id*. Väli *request_id* toimib unikaalse identifikaatorina, mis seob kõik ühe kasutaja tegevusega seotud logid omavahel, sõltumata sellest, millises teenuses need tekkisid. Väli *event_name* annab täiendavat konteksti, kirjeldades praeguse päringu olemust.

Kuna teenused suhtlevad omavahel võrgu kaudu, tekkis vajadus neid identifikaatoreid teenuste vahel edastada. Lahendusena rakendasime andmeedastuse HTTP päiste (ingl. k *headers*) kaudu. Protsess toimib järgmiselt: kui kasutaja toiming algatab süsteemis uue päringu, genereerib vastuvõttev teenus unikaalse *request_id* ja määrab *event_name* väärtuse. Need andmed lisatakse kõigi väljaminevate päringute HTTP päistesse.

Kui mõni teine süsteemi teenus võtab selle päringu vastu, võtab ta esimese asjana päistest *request_id* ja *event_name* väärtused. Seejärel lisab teenuse sisene logimislahendus need identifikaatorid automaatselt igale logikirjele, mis luuakse antud päringu töötlemise elutsükli jooksul.

See lahendus võimaldab Grafana keskkonna arendajatel logisid filtreerida ühe päringu alusel. See annab üksikasjaliku ja kronoloogilise ülevaate kogu päringu teekonnast läbi erinevate teenuste, kiirendades ja parandades oluliselt kitsaskohtade tuvastamise tõhusust.

4.5 Logimise juurutamine ja logimistasemete kasutamine

Süsteemi uuele seirearhitektuurile migreerimisel oli oluline asendada vanad ja ebaefektiivsed logimismeetodid. Eelmine süsteem, mis tugines peamiselt standardiseerimata väljundile, eemaldasime täielikult. Selle asemel integreerisime kõigisse süsteemiklassidesse uuele arhitektuurile loodud logimisklass. See muudatus tagas, et kõik sündmused läbivad ühe töötlusprotsessi ja jõuavad kesksserverisse standardiseeritud vormingus.

Süsteemi jälgitavuse parandamiseks ja teabe loetavuse tagamiseks rakendasime selge eristus ja loogika nelja logimistaseme vahel, mis on kirjeldatud tabelis 1.

Tabel 1. Logimistasemed ja nende kasutus.

Tase	**Eesmärk**	**Näide**
INFO	Teenuse oluliste tegevuste alguse ja lõpu märkimiseks. Annab kiire ülevaate süsteemi üldisest töövoost.	Päringu vastuvõtmine, tegevuse edukas lõpetamine
DEBUG	Päringute elutsükli detailseks jälgimiseks ja vahesammude tuvastamiseks. Ei sisalda suuri või tundlikke andmehulki.	Andmebaasi edukas otsingud
WARNING	Oodatud, kuid ebatavalised olukorrad, mis ei katkesta süsteemi tööd, kuid vajavad tähelepanu.	Autoriseerimisvead, ebaõnnestunud andmebaasi otsingud
ERROR	Ootamatud süsteemivead ja kriitilised erandid, mille tagajärjel tagastatakse kasutajale kood 500. Sisaldab lisaks veateatele ka täielikku pinu jälge (ingl. k *stack trace*).	Päringu töötlemisel tekkis serveri sisemine viga

Eelmises peatükis kirjeldatud logistruktuuri *data* väli mängib üliolulist rolli sündmusele vajaliku konteksti andmisel. See JSON-objekt lisab dünaamiliselt konkreetse päringuga seotud andmeid, näiteks Kooudi, kursuse üliõpilaste identifikaatoreid või nimesid. See andmestruktuur võimaldab logisid Grafanas hiljem filtreerida konkreetse objekti või isiku alusel. Andmeturve ja privaatsus on hajutatud süsteemides olulised, seega rakendasime ranged kontrolli: tagasime käsitsi, et tundlikke andmeid, näiteks paroole või juurdepääsutõendid, ei lisataks logidesse.

4.6 Mõõdikute kogumise integratsioon

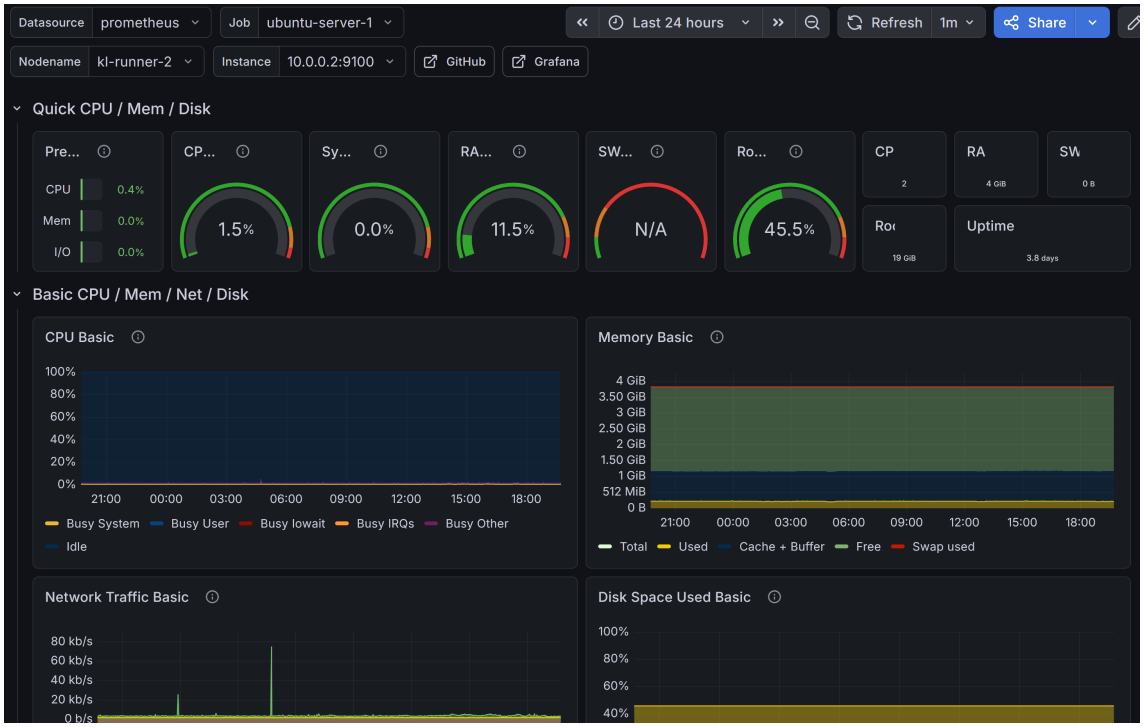
Pärast Mimir'i ja Prometheuse paigaldamist serverisse seadistasime monitooringu töövoogu nii, et Prometheus kogub teenuste poolt spetsiaalsetest lõpp-punktidest eksponeeritud meetrikaid ja edastab need Mimir'isse pikaajaliseks salvestamiseks. Seejärel kasutab Grafana Mimir'it andmeallikana nende meetrikate visualiseerimiseks töölaudadel. Otsustasime

koguda meetrikaid ainult teenustelt Kooud tagarakendus, *Request Forwarder* ja *Test Runner*.

Teenuse Kooud tagarakendus puhul kasutasime meetrikate kogumiseks paketti *prometheus-net*. See pakett võimaldab Prometheuse integratsiooni .NET rakendustega, pakkudes lihtsat võimalust standardsete rakendusmõõdikute, näiteks CPU kasutuse, kogumiseks ning ka kohandatud meetrikate loomiseks. Rakendasime kohandatud meetrika nimega *charon_tester_callback_requests_total*, mis loendab esitatud päringute koguarvu ning sisaldab silte *course* ja *kooud*. Need sildid võimaldavad meetrika filtreerimist ja agregeerimist Grafana töölaudadel [39].

Ülejäänud teenuste puhul otsustasime koguda mõõdikuid serveri tasandil, mitte otse rakendustest, kuna rakendusepõhiseid kohandatud töölaudu ei olnud vaja ning iga server majutab ainult ühte rakendust. Selleks kasutasime Node Exporterit, mis on iseseisev mõõdikute kogumise agent ning kogub OS (*Operation System*) mõõdikuid ja eksponeerib need Prometheuse formaadis [40].

Nagu on näidatud joonisel 9, pakub Grafana Node Exporteri jaoks eelnevalt loodud töölauda, mis sisaldab kõiki vajalikke paneele serveri ressursikasutuse jälgimiseks, näiteks CPU, mälu, ketta ja võrgu kasutuse monitoorimiseks.



Joonis 9. Node Expoter eeltehtud töölaud.

4.7 Keskkondade seadistus

Kooud ja Arete teenuste Kubernetesele migreerimiseks seadistati kaks eraldiseisvat keskkonda - arenduskeskkond lokaalseks testimiseks ja testimiskeskkond GitLabi platvormil.

4.7.1 Arenduskeskkond

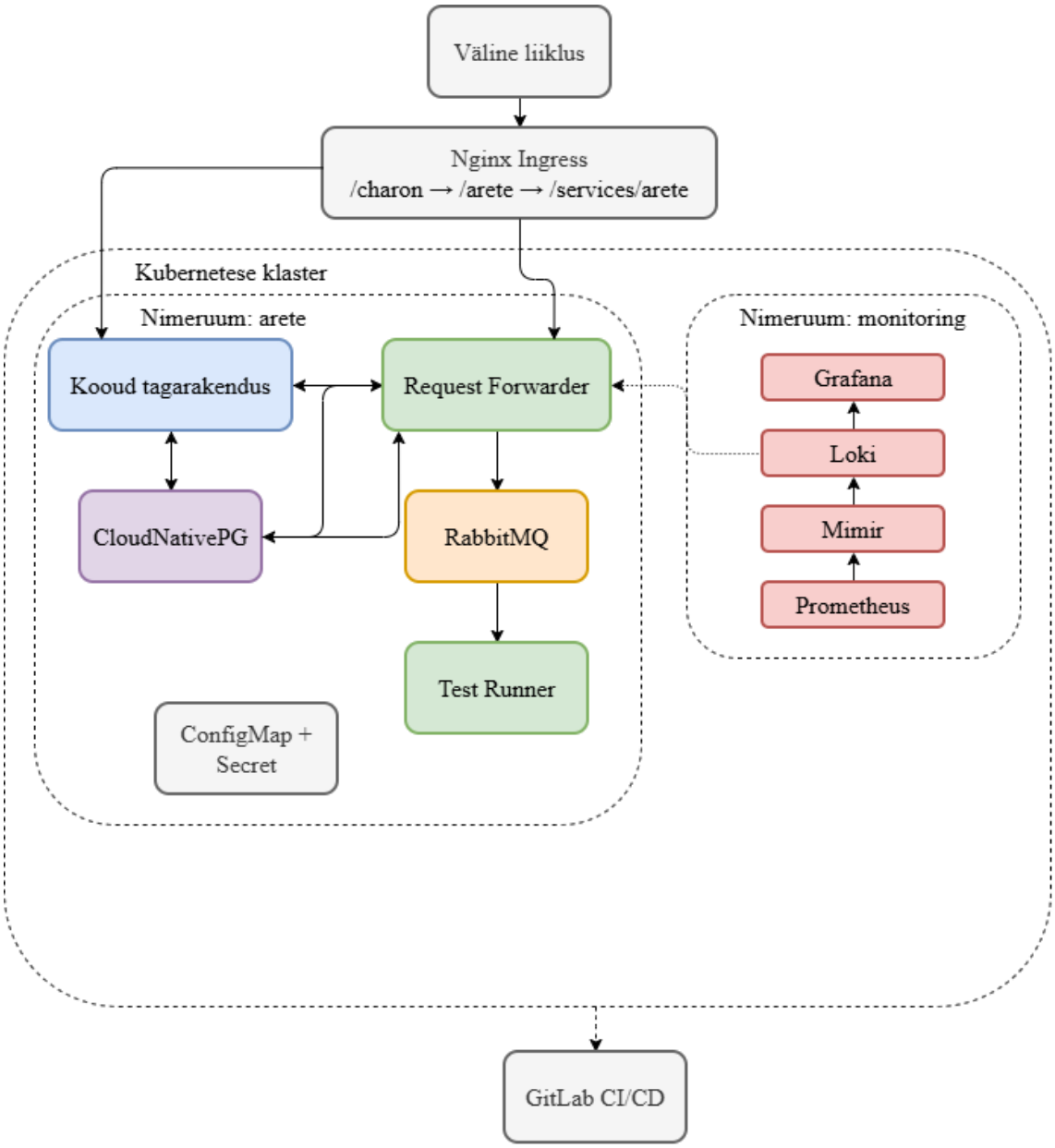
Lokaalse arenduskeskkonna seadistamiseks kasutati Podman Desktopi [41] koos Kind tööriistaga [42]. Podman on avatud lähtekoodiga konteinerite haldamise tööriist, mis toimib alternatiivina Dockerile [41]. Podmani peamine eelis seisneb daemonivaba arhitektuuris - erinevalt Dockerist ei vaja Podman taustalt töötavat privilegeeritud protsessi, mis muudab selle turvalisemaks. Kind (ingl. k *Kubernetes in Docker*) on Podman Desktopi tööriist, mis võimaldab käivitada Kubernetese klastri lokaalselt Podman-konteinerite sees [42]. See lahendus sobib ideaalselt arenduskeskkonnaks, kuna võimaldab testida Kubernetese konfiguratsioone lokaalselt ilma eraldi tootmisserverita. Käesolevas töös seadistati Kind ühe sõlmega klastrina Podman Desktopi kaudu, mis võimaldas kõigi Kubernetese ressursside - juurutuste, teenuste, Ingressi ja püsiva salvestusmahu taotluste - lokaalset testimist enne testimiskeskkonda viimist.

4.7.2 Testimiskeskkond

Arenduskeskkonnana kasutati lokaalset Kubernetese klastrit Kind tööriistaga Podman Desktopi kaudu. Testimiskeskkonnana kasutati DigitalOcean haldatavat Kubernetese teenust (DOKS) [43].

4.8 Kubernetese arhitektuursed otsused

Käesolevas alapeatükis kirjeldatakse peamisi arhitektuurseid otsuseid, mis tehakse Kooud ja Arete teenuste Kubernetesele migreerimisel. Joonis 10 näitab Kooud ja Arete teenuste Kubernetese arhitektuuri.



Joonis 10. Kooud ja Arete teenuste Kubernetese arhitektuur.

4.8.1 Migratsioonistrateegia

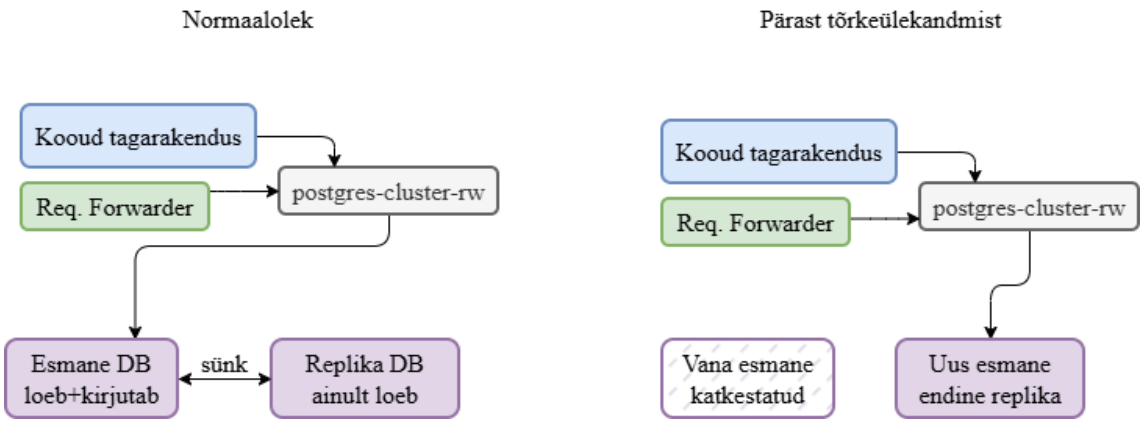
Kooud ja Arete teenuste migreerimine Kubernetese platvormile toimus järk-järgult. Enne Kubernetesele üleminekut kirjutasime olemasolevad teenuste konfiguratsioonid ümber Podman Quadlet failideks, mis on `systemd`-põhised süsteemsete teenustena hallatavad konteinerimääratlused. See samm võimaldas säilitada olemasoleva funktsionaalsuse, standardiseerides samal ajal teenuste käivitamise ja haldamise. Kubernetese konfiguratsioonid arendasime ja testisime paralleelselt, võimaldades sujuvat üleminekut ilma teenuste katkestuseta. Migratsioon järgis põhimõtet, kus iga teenus viidi Kubernetesesse iseseisvalt ning

valideeriti enne järgmise teenuse migreerimist. Nimeruumi konfiguratsioon on esitatud lisas 6.

4.8.2 Andmebaasihaldus

Andmebaasihalduseks valisime CloudNativePG operaatori, mis on spetsiaalselt PostgreSQL klastrite haldamiseks loodud Kubernetese laiendus [19].

Tavalise PostgreSQL juurutuse asemel pakub CloudNativePG mitmeid olulisi eeliseid. Esiteks toetab see automaatset tõrkeülekandumist - kui esmane andmebaasisõlm lakkab töötamast, ülendatakse replika automaatselt uueks esmaseks sõlmeks ilma käsitsi sekkumiseta. Teiseks on sisseehitatud replikatsioon, mis tagab andmete kõrge käideldavuse. Kolmandaks võimaldab üks CloudNativePG klaster hallata mitut andmebaasi korraga - käesolevas töös katab üks klaster nii Arete kui ka Kooudi andmebaaside vajadused, mis lihtsustab haldust ja vähendab ressursikasutust. Joonis 11 illustreerib CloudNativePG automaatset tõrkeülekandmist.



Joonis 11. CloudNativePG automaatne tõrkeülekandmine.

4.8.3 Liikluse marsruutimine

Välise HTTP-liikluse marsruutimiseks valisime Nginx Ingressi, mis paigaldatakse Helmi kaudu. Nginx on üks enim kasutatavaid ja tõestatud veebiserveri ning pöördproksi lahendusi, millel on ulatuslik dokumentatsioon ja kogukonna tugi. Nginx Ingress võimaldab määratleda URLide põhiseid marsruutimisreegleid deklaratiivselt Kubernetese ressurssidena. Seadistasime marsruutimisreeglid nii, et `/charon` päringud suunatakse `charon-backend` teenusele ja `/arete` päringud `request-forwarder` teenusele [44].

4.8.4 Konfiguratsioonide ja saladuste haldus

Konfiguratsiooniväärtuste ja tundlike andmete haldamiseks kasutasime Kubernetese natiivseid lahendusi - seadistuskaarti ja saladust. Seadistuskaart sisaldab avalikke seadeid nagu andmebaasi aadress, sõnumivahendaja hostinimi ja logimistase. Saladus hoiab tundlikke andmeid nagu andmebaasiparoolid, API võtmed, GitLabi registri juurdepääsuvõtmed ja SSH-võtmed, eraldades need konteineripiltidest ja lähtekoodihoidlast. Mõlemad ressursid süstisime konteineritesse keskkonnamuutujatena, mis tagab, et tundlikud andmed ei ole konteineripiltidesse sisse kodeeritud ega lähtekoodihoidlasse salvestatud. See lähenemine järgib turvalisuse parimat praktikat, eraldades rakenduse konfiguratsiooni selle koodist.

4.9 Teenuste migreerimine Kubernetese platvormile

Käesolev peatükk kirjeldab Kooud ja Arete teenuste migreerimise tehnilist teostust. Migreerimine toimus kahes järjestikuses etapis: esmalt kirjutati olemasolevad teenuste konfiguratsioonid ümber Podman Quadlet failideks, seejärel arendati ja rakendati Kubernetese konfiguratsioonid. See lähenemine võimaldas säilitada teenuste toimimise migreerimise ajal ning valideerida iga muudatus enne järgmise etapi alustamist.

4.9.1 Podman Quadlet failide seadistamine

Migreerimise esimese etapina asendati olemasolevad teenuste konfiguratsioonid Podman Quadlet failidega, mis võimaldab hallata konteinereid systemd teenustena - konteinerid käivituvad automaatselt koos serveriga ning operatsioonisüsteem vastutab nende elutsükli haldamise eest [45]. Iga teenus kirjeldati eraldiseisva Quadlet failina, kus teenuste omavahelised sõltuvused määratleti `Requires` ja `After` direktiividega. Need direktiivid tagavad õige käivitusjärjekorra - näiteks *Request Forwarder* teenus käivitub alles pärast seda, kui andmebaas ja sõnumivahendaja on käivitunud. Keskkonnamuutujad loeti tsentraalsest `/etc/arete/.env` failist, mis eraldas konfiguratsioonid konteinerimääratlustest ja tagas, et tundlikud andmed ei ole failidesse sisse kodeeritud. Kõik Podman Quadlet konfiguratsioonifailid on esitatud lisas 7.

4.9.2 Andmebaasi seadistamine

CloudNativePG klaster seadistati kahe eksemplariga - üks esmane ja üks replika - mis sisaldab ühist andmebaasi nii Kooudi kui ka *request-forwarderi* teenuse jaoks. Andmebaasi algseadistus teostatakse automaatselt klastri loomisel - CloudNativePG loob määratud andmebaasi ja kasutaja ning rakendab juurdepääsuõigused. Andmebaasi juurdepääsuvõtmed hoitakse Kubernetese saladusena ning süstitakse konteineritesse keskkonnamuutujatena. Andmebaasiga ühenduse loomiseks kasutavad teenused klastri sisest DNS-nime `postgres-cluster-rw`, mis osutab alati esmasele andmebaasisõlmele. Replika tõrkeülekandumise korral uuendab CloudNativePG DNS-i automaatselt nii, et teenused ei pea teadma, milline füüsiline sõlm on parasjagu esmane. Andmebaaside initsialiseerimine - tabelite loomine ja algandmete lisamine - teostati eraldi Kubernetese töö (ingl. k *job*) kaudu, mis käivitati pärast klastri loomist [12].

4.9.3 Sõnumivahendaja seadistamine

Sõnumivahendajana kasutatakse RabbitMQ-d, mis paigaldati Kubernetese juurutusena koos püsiva salvestusmahu taotlusega. Püsiv salvestusruum tagab, et sõnumijärjekorras olevad esitused säilivad ka podi taaskäivituse korral - see lahendab otseselt peatükis 2.1 kirjeldatud probleemi, kus RabbitMQ taaskäivituse korral kadusid kõik järjekorras olevad esitused [46].

Varasem probleem, kus RabbitMQ sõnumijärjekorda saadeti koos päringuga ka failid ja metaandmed, ületades limiidi, lahendatakse Garage S3 objektisalvestuse integreerimisega, mis on planeeritud edasise arendustöö osana.

RabbitMQ seisundi jälgimiseks kasutati elavuse- ja valmisolekuproove, mis käivitavad RabbitMQ diagnostikakäsklusi otse konteineri sees. Elavusproov käivitab käskluse `rabbitmq-diagnostics ping`, mis kontrollib, kas RabbitMQ protsess vastab. Valmisolekuproov käivitab käskluse `rabbitmq-diagnostics check-running`, mis kontrollib, kas RabbitMQ on päringute vastuvõtmiseks valmis. See eristus on oluline - teenus võib olla käivitunud, kuid mitte veel valmis päringuid vastu võtma. Elavuse- ja valmisolekuproovid lisati kõigile juurutustele. RabbitMQ konfiguratsioonifailid on esitatud lisas 8.

4.9.4 Teenuste juurutamine Kuberneteses

Kõik kolm põhiteenust - Kooud tagarakendus, `request-forwarder` ja `test-runner` - juurutati eraldiseisvate Kubernetese juurutustena nimeruumi `arete`. Iga juurutus koosneb kolmest ressursist: juurutusest, teenusest ja vajadusel püsiva salvestusmahu taotlusest.

Teenuste käivitusjärjekorra tagamiseks kasutati initsialiseerimisakonteinereid, mis kontrollivad perioodiliselt, kas sõltuvusteenused on kättesaadavad. *Request Forwarder* ootab nii andmebaasi kui ka sõnumivahendaja valmidust. *Kooud Backend* ootab ainult andmebaasi valmidust. *Test Runner* ootab ainult sõnumivahendajat.

Kõikidele juurutustele lisati elavuse- ja valmisolekuproovid, mis saadavad perioodiliselt päringu iga teenuse seisundikontrolli otspunktile. `request-forwarder` kasutab otspunkti `/services/arete/api/v2/health`, `kooud-backend` kasutab otspunkti `/health` ja `test-runner` kasutab Spring Boot aktuaatori otspunkti `/actuator/health` [47]. Valmisolekuproov käivitub 45 sekundi pärast ja elavusproov 60 sekundi pärast konteineri käivitumist, mis annab teenustele piisavalt aega käivitumiseks.

GitLabi registrist konteineripiltide tõmbamiseks seadistati kõikidele juurutustele GitLabi registri juurdepääsuvõtmed Kubernetese saladusena. See võimaldab Kubernetisel automaatselt tõmmata uusimaid konteineripilte ilma käsitsi sekkumiseta.

Request Forwarderi konfiguratsioonifailid on esitatud lisas 9 ning Test Runneri konfiguratsioonifailid lisas 10.

4.9.5 Liikluse marsruutimine

Välise liikluse marsruutimiseks seadistati Nginx Ingress marsruutimisreeglitega, mis suunavad päringud õigetele teenustele URLi põhjal. Marsruutimisreeglid on järgmised: `/arete` päringud suunatakse *Request Forwarder* teenusele ja `/charon` päringud `kooud-backend` teenusele [44].

Nginx Ingressile seadistati pikemad päringu maksimaalsed kestused - 3600 sekundit nii lugemiseks kui ka saatmiseks - kuna koodiesituste testimine, eriti masinõppe mudelite puhul, võib kesta kaua. Failide üleslaadimise piirmääraks seadistati 50 megabaiti, mis katab tudengite koodifailide ja testide suuruse. URLi ümberkirjutamine (ingl. k *rewrite*) seadistati

nii, et teenused saavad päringuid ilma eesliiteta - näiteks `/arete/api/test` päring jõuab teenuseni kui `/api/test`.

5 Tulemused

See peatükk hindab väljatöötatud tehnoloogiliste lahenduste edukust ja rakendatavust. See hõlmab uue arhitektuuri tehnilist valideerimist ja koormustestimist, tulemuste analüüsi ning soovitusi süsteemi edasiseks arendamiseks.

5.1 Valideerimine

Käesolevas alapeatükk annab ülevaate loodud lahenduste valideerimisprotsessist. Esmalt kirjeldame uue seiresüsteemi valideerimist tehniliste testide ja kasutajakogemuse analüüsi kaudu. Seejärel kirjeldame Kubernetese konfiguratsioonide valideerimist testimiskeskkonnas, kus kontrollisime teenuste edukat käivitumist, omavahelist suhtlust ning automaatse taastumise mehhanismi toimimist.

5.1.1 Seiresüsteemi valideerimine

Uue seirelahenduse valideerimisel kasutasime tehnilisi teste ja kasutajakogemuse analüüsi. Tehniliste testide käigus saatsime süsteemile mitu käsitsi päringut, et kontrollida andmeedastuse kiirus ja terviklikkus. Testid kinnitasid, et kõik teenustest saadetud logid jõudsid edukalt Grafana keskkonda ilma viivituste ja andmete kadumiseta. Erilist tähelepanu pöörasime logiahelale: testimine näitas, et unikaalse *request_id* abil saime edukalt jälgida koodi esitamise teed läbi kõigi süsteemikomponentide, alates saatmisest kuni tulemuste salvestamiseni.

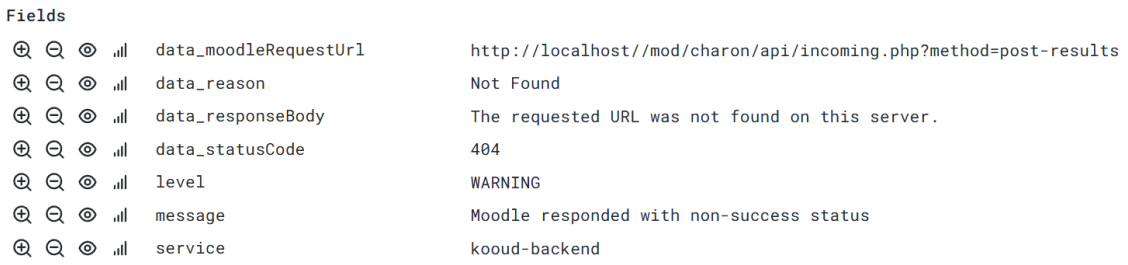
Lisaks viisime mitme arendajaga läbi võrdlevad testid, et hinnata, kui kiiresti suudetakse vead üles leida vanas süsteemis (ilma seireta) ja uues lahenduses. Testisime reaalset olukorda, kus esitlusel oli olematu testimistüüp. Tänu sellele, et uues lahenduses asuvad kõik logid ühes kohas ja kasutusel on *request_id*, kulus probleemi tuvastamiseks vaid 2 minutit, samas kui vanas süsteemis võttis sama vea leidmine aega 15 minutit. Erinevate arendajate puhul jäi ajaline võit uue süsteemi kasuks alati 7–8-kordseks.

Mõõdikute kogumiseks ja visualiseerimiseks loodi terviklik seirelahendus, mis võimaldab arendajatel süsteemi tööd mugavalt jälgida Grafana töölaudade kaudu. Selle raames valmis Kooudi tagarakenduse töölaud, mis võimaldab reaalajas jälgida esituste arvu ning filtreerida andmeid kursuse ja Kooudi alusel. Lisaks on võimalik kuvada hetkel kõige populaarsemaid kursusi ja ülesandeid (vt lisa 11).

Samuti on saadaval Kooudi tagarakenduse süsteemimõõdikud, sealhulgas protsessori kasutuskoormus ning andmebaasi ühenduste arv (vt lisa 12). Teiste serverite ja teenuste jälgimiseks paigaldati ja seadistati Node Exporter, mis kogub mõõdikuid neljast serverist. See võimaldab kasutada Grafanas Node Exporteri töölauda, mille kaudu saab jälgida serverite ressursikasutust ja tehnilist seisundit (vt lisa 13).

Loodud logimislahenduse praktiline väärtus leidis tõestust juba arendus- ja testimisfaasis, aidates tuvastada vea, mis varasemalt oli jäänud märkamatuks. Süsteem peab pärast tudengi koodiesituse edukat testimist saatma Moodle'i keskkonda päringu, et uuendada vastava programmeerimisülesande hindeid. Uued Grafana logid näitasid, et uuendatud tulemused ei jõudnud tegelikult kunagi Moodle'isse.

Nagu näha jooniselt 12, salvestas süsteem *WARNING*-taseme logi teatega *"Moodle responded with non-success status"*. Struktuursest *data* väljast eraldatud parameetrid näitasid koheselt, et probleem peitus vigases sihtkoha võrguaadressis. Eelmises infrastruktuuris oleks selline andmekadu mõnda aega märkamata jäänud.



Joonis 12. Moodle'i sünkroniseerimise vea tuvastamine käsitsi testimise käigus Grafana logide abil.

Lisaks tehnilisele ülevaatele kogusime tagasisidet süsteemi peamistelt kasutajatelt: arendajatelt ja abiõppejõududelt. Kasutajad märkisid, et uus lahendus on oluliselt põhjalikum ning võimaldab vigade ja hoiatuste tuvastamist reaalajas, kiirendades probleemide lahendamist. Samuti tõstsid nad esile riistvararessursside (protsessor, mälu) ja äriloogika mõõdikute

(koodiesituste arv) jälgimise lihtsust. Erinevalt eelmisest lahendusest, kus graafikud olid staatilised, pakub Grafana Stack võimalust muuta ajaraamid ja kohandada töölaudu vastavalt vajadusele, suurendades oluliselt analüüsiprotsessi paindlikkust. Saadud töölauad osutusid kasutajatele intuitiivseteks, kinnitades süsteemi kasutusmugavust.

5.1.2 Kubernetese konfiguratsioonide ja jõudluse valideerimine

Kubernetese konfiguratsioonide valideerimiseks juurutati kõik teenused DigitalOcean haldatavasse Kubernetese klastrisse [43], mis koosnes kolmest töösõlmest, mis jaotati erinevatesse kättesaadavustsoonidesse(vt lisa 14). DigitalOcean valiti testimisplatvormiks, kuna ülikooli oma klaster ei ole veel seadistatud ning teenuste koodibaas ei ole tootmiskeskkonnaks valmis - Kooud tagarakendus on praegu Moodle'i pistikprogrammina juurutatud ning eraldiseisva teenusena ümberkirjutamine on alles planeerimisjärgus, mistõttu ei olnud võimalik testida täielikku tootmiskeskkonda. Testimise peamine eesmärk oli kontrollida Kubernetese konfiguratsioonide õigsust ning veenduda, et klastri seadistamine on lihtne ja kiire. Konfiguratsioonid rakendati käsitsi `kubectl` käskude abil. GitLabi Kubernetese agendi kasutamine oli planeeritud, kuid meie GitLabi versiooni ja Kubernetese ühilduvusprobleemide tõttu loobuti sellest testimisfaasis - see ei olnud testimise eesmärgi seisukohalt oluline. GitLabi agendi integratsioon on planeeritud edasise arendustöö osana.

Testimise käigus kontrolliti kõigi teenuste käivitumist ja omavahelist suhtlust. Valideeriti, et initsialiseerimisakonteinerid tagavad õige käivitusjärjekorra - teenused käivitusid alles pärast kõigi sõltuvuste valmidust. Elavuse- ja valmisolekuproovid töötasid ootuspäraselt: Kubernetes tuvastas automaatselt, millal teenus oli päringute vastuvõtmiseks valmis, ning suunas liikluse õigetele teenustele. Nginx Ingressi marsruutimisreeglid valideeriti, saates päringuid erinevatele URLidele ja kontrollides, et need jõudsid õigetele teenustele.Kubernetese klastri ülevaate töölaud koos nimeruumide statistikaga on esitatud lisas 16. Kogu klastri esmakordne seadistamine ja teenuste juurutamine võttis aega ligikaudu üks tund - see on märkimisväärne paranemine võrreldes varasema intsidendiga, kus tulemüüri vale seadistuse tõttu kulus teenuste taastamiseks 12-14 tundi ning probleemi asukoht ei olnud selge.

Funktsionaalne End-to-End test

Pärast kõigi komponentide juurutamist viidi läbi funktsionaalne valideerimine täistsükli testimisega. Test teostati saates HTTP POST päring otspunktile `/arete/submission/test` - lihtne Pythoni programm, mis loeb arvu ja väljastab selle kahekordse väärtuse. Sisendandmed: `5`, oodatav tulemus: `10`. Süsteem tagastas `"failed": false`, mis kinnitas kogu töötlusahela korrektsuse: HTTP päring → Nginx Ingress → Request Forwarder → RabbitMQ → Test Runner → testija konteiner → tulemus. Arete nimeruumi teenuste logid testimise ajal on esitatud lisas 18.

Podi taastumise test

Automaatse taastumise mehhanismi testimiseks kustutati Test Runneri pod käsitsi käsuga `kubectl delete pod`. Kubernetes ReplicaSet kontroller tuvastas podi puudumise ühe sekundi jooksul ja lõi automaatselt uue podi. Ligikaudu 25 sekundi pärast olid mõlemad konteinerid - nii Test Runner kui ka Docker-in-Docker kaasnev konteiner - täielikult käivitunud ja valmis päringuid vastu võtma. Kogu taastumisprotsess toimus ilma igasuguse käsitsi sekkumiseta, kinnitades *self-healing* mehhanismi töökindlust.

Koormustestimine k6 tööriistaga

Koormustestimiseks kasutati k6 tööriista [48] kahe järjestikuse stsenaariumiga. Baseline stsenaariumis simuleerisid 5 virtuaalset kasutajat (ingl. k *virtual user*) samaaegselt koodiesituste saatmist 60 sekundi jooksul - iga virtuaalne kasutaja saatis uue päringu kohe pärast eelmise vastuse saamist. Stress stsenaariumis kasvas virtuaalsete kasutajate arv järk-järgult 5-lt 50-le ning püsis 50 kasutajal 2 minutit. Kokku sooritati 1491 iteratsiooni. Kuni 20-25 samaaegse kasutaja juures töötas süsteem stabiilselt ning vastamisajad jäid piiridesse. Üle 25 kasutaja puhul ületas osa päringute vastamisaeg seatud piiri 10 sekundit - RabbitMQ järjekorda hakkas kogunema ülesandeid kiiremini kui üksik *Test Runner* jõuab töödelda (umbes 0,4 päringut sekundis). Oluline on see, et ühegi teenuse puhul ei esinenud kokkujooksmisi ega taaskäivitusi - tekkis kontrollitud jõudluse langus. RabbitMQ järjekord ei tühjenud testimise jooksul, kuna testimiskeskkonnas piirduti ühe *Test Runneri* eksemplariga. Oma klastris oleks lahenduseks replikate arvu suurendamine, mis võimaldab paralleelselt töödelda mitu ülesannet korraga - näiteks 20-30 replikat kataks 70 samaaegse kasutaja koormuse. Koormustestimise ajal jäi klastri ressursikasutus mõistlikes piirides (vt lisa 15).

Eksamisituatsiooni simulatsioon

Reaalse eksamisituatsiooni simuleerimiseks viidi läbi kahefaasiline test 70 samaaegse virtuaalse kasutajaga. Esimeses faasis (asünkroonne, 10 minutit) saadeti 6724 päringut, millest 99,9% võeti vastu edukalt (HTTP 202) läbilaskevõimega 8,3 päringut sekundis. Nginx Ingress ja Request Forwarder tulid 70 paralleelse kasutajaga suurepäraselt toime. Teises faasis (sünkroonne, 3 minutit) selgus süsteemi kitsaskoht: 10 minuti jooksul kogunenud ligikaudu 4700 ülesannet olid RabbitMQ järjekorras ning sünkroonsed päringud aegusid 120 sekundi pärast, oodates töötlemist. Oluline on märkida, et esitused ei läinud kaotsi - need jäid RabbitMQ järjekorda ning töödeldakse järjest. Kasutaja vaatepunktist tähendas aegumine seda, et ta ei saanud kohest vastust, kuid süsteemi tasandil säilis iga esitus. Ajalõpu pikendamine leevendaks seda probleemi kasutajakogemuse seisukohalt, kuid põhiline lahendus on *Test Runneri* replikate arvu suurendamine, mis vähendab järjekorras ootamise aega. Üksiku *Test Runneri* läbilaskevõime on ligikaudu 0,4 päringut sekundis, mistõttu vajab 70 kasutaja teenindamiseks ligikaudu 20-30 *Test Runneri* replikat.

Kriitilise koormuse analüüs

Süsteemi kriitilise koormuse punkti leidmiseks viidi läbi järk-järgult kasvava koormusega test kuni 150 samaaegse kasutajani. Tulemused on esitatud tabelis 2.

Tabel 2. Kriitilise koormuse analüüsi tulemused.

Kasutajaid	**Vastuvõetud**	**Käitumine**
10	100%	Stabiilne, avg < 500 ms
30	100%	Stabiilne
60	100%	Järjekord hakkab kasvama
100	~85%	Aegumine 10 sekundi piiril
150	69,2%	Kriitilise koormuse punkt

Kokku võeti vastu 3477 päringut ning 1545 päringut aegusid - need jõudsid küll Request Forwarderini, kuid ootasid töötlemist kauem kui lubatud ajalõpp. Oluline on märkida, et kriitilise koormuse punkti saabudes ei langenud süsteem täielikult - RabbitMQ puhverdas järjekorda ning Kubernetes jätkas töötamist. See on põhimõtteline erinevus võrreldes varasema lahendusega, kus päringute puhverdamine puudus ning koormuse kasvades aeglustus süsteem kontrollimatult ilma võimaluseta automaatselt taastuda. Kolme töösõlme

ressursikasutuse detailvaade koormustestimise ajal on esitatud lisas 17.

Suure sisendi test

Eraldi testiti süsteemi käitumist 150 MB suuruse päringuga, et kontrollida veatingimustes käitumist. Testimisel avastati kolm järjestikust piirangut. Esiteks Nginx Ingressi vaikimisi keha suuruse limiit 50 MB, mis lahendati tõstes limiiti 200 MB-ni. Teiseks RabbitMQ maksimaalse sõnumi suuruse limiit 128 MB, mis põhjustas sõnumi tagasilükkamise. Kolmandaks Jackson andmeteegi [49] - laialdaselt kasutatav Java JSON töötlemise teek - vaikimisi JSON-stringi suuruse limiit 20 MB, mis ei ole praegu lahendatud. Kõigi kolme piirangu õige arhitektuurne lahendus oleks Garage objektisalvestuse integreerimine - läbi RabbitMQ edastataks ainult viide salvestatud failile, mitte faili sisu, mis kõrvaldaks kõik kolm piirangut korraga. Selleni käesoleva töö raames ei jõutud ning see on planeeritud edasise arendustöö osana.

Andmebaasi jõudluse test

CloudNativePG klastri jõudluse kontrollimiseks kasutati pgbench tööriista [50] - PostgreSQL-i sisseehitatud koormustestimise vahendit. Testi aluseks võeti TPC-B standard [51], mis simuleerib pangandussüsteemile omast töökoormust, kus iga tehingu käigus tehakse SELECT, UPDATE ja INSERT operatsioone. Mõõduka koormusega (10 klienti) saavutas esmane sõlm 200 tehingut sekundis keskmise latentsusega 50 ms ning vigadeta. Suure koormusega (50 klienti) tõusis läbilaskevõime 233 tehinguni sekundis, kuid latentsus kasvas 213 ms-ni. Latentsuse kasv on PostgreSQL-le tüüpiline - suurema koormusega tekib ridade lukustamisel ootejärjekord, mis suurendab keskmist vastamisaega [52]. Replika sõlm saavutas ainult lugemisoperatsioonidega 3304 tehingut sekundis - 14 korda kiiremini kui esmane sõlm samal koormusel, kuna lugemisoperatsioonidel puuduvad lukustuskonfliktid. Kõigis kolmes testis oli vigade arv null ja replikatsiooni viivitus püsis kogu testimise vältel 0 baiti, mis kinnitab CloudNativePG sünkroonse replikatsiooni töökindlust.

Kokkuvõte

Testitulemuste kokkuvõte on esitatud tabelis 3. Kuni 70 samaaegse kasutajaga (reaalne eksamisituatsioon) võtab süsteem vastu 99,9% päringuid vigadeta. Kriitilise koormuse punkt saabub 100-150 kasutaja juures - süsteem ei lange täielikult, kuna RabbitMQ puhverdab järjekorda ning Kubernetes taastab langenud komponendid automaatselt ligikaudu 25 sekundi jooksul. Suuremamahulise testimise mehhanismid jäid käesoleva töö skoobist

välja. Edasise skaleerimise jaoks piisab Test Runneri replikate arvu suurendamisest või horisontaalse automaatse skaleerimise (HPA) seadistamisest RabbitMQ järjekorra pikkuse põhjal.

Tabel 3. Valideerimistestide kokkuvõte.

Test	**Tulemus**	**Kitsaskoht**	**Lahendus**
Baseline (5 kasutajat)	Läbitud	—	—
Stress (50 kasutajat)	Aegumine	Üksik Test Runner	Skaleerimine
Eksam async (70 kasutajat)	Läbitud	—	—
Eksam sync	Aegumine	Järjekord täis	Skaleerimine
Kriitilise koormus (150 kasutajat)	Osaline tõrge	Üksik Test Runner	Skaleerimine
150 MB sisend	Ebaõnnestus	3 piirangut	Garage
Podi taastumine	Läbitud	—	~25 sek
PostgreSQL koormus	Läbitud	—	—

5.2 Tulemuste analüüs

Töö tulemusena täitsime kõik arendusetapis püstitatud nõuded. Uurisime olemasolevat tehnoloogiate turgu ning valisime projekti jaoks sobivaimad lahendused - Grafana Stack ja Kubernetes. Projekti käigus paigaldasime ja seadistasime edukalt kogu vajalik Grafana tarkvarakomplekt, sealhulgas Grafana, Loki, Mimir ja Prometheus. Lisaks lõime ühtne logimisstandard, mida kasutatakse kõikides Kooudi ja Arete teenustes ning mis on täielikult ühilduv Grafana Loki logihalduslahendusega.

Töö käigus kavandasime ja realiseerisime seireandmete kogumise arhitektuur ning määratlesime andmevood erinevate süsteemikomponentide vahel. Rakenduste koodibaasi integreerisime Loki logikogumise lahendus, mille tulemusena saadetakse enamiku teenuste logid Loki süsteemi iga viie sekundi järel.

Moodle teenuse puhul esinesid tehnilised piirangud. Logide regulaarne edastamine on seal piiratud minimaalselt ühe minuti intervalliga, samas kui teistes teenustes toimub logide edastus kuni iga viie sekundi järel. Seetõttu ei ole võimalik tagada sama tihedat

logivoogu. Lisaks puudub Moodle keskkonnal Prometheuse integratsioon, millepärast puudub võimalus koguda ega edastada mõõdikuid Prometheuse süsteemi kaudu. Tegemist on Moodle'i platvormi piirangutega, mille haldamine ja arendus on eraldi Moodle'i meeskonna vastutusel.

Samuti seadistasime mõõdikute kogumine Prometheuse abil ning lõime kasutajasõbralikud töölauad logide ja mõõdikute visualiseerimiseks. Valminud lahendus annab kasutajatele keskse keskkonna, kus on võimalik jälgida kogu süsteemi logisid ja mõõdikuid reaalajas. See parandab süsteemi läbipaistvust, lihtsustab probleemide tuvastamist ning toetab süsteemi töökindluse ja hooldatavuse parandamist.

Kubernetese platvormile migreerimise käigus kavandasime ja realiseerisime klastri arhitektuuri koos nimeruumide, juurutuste, teenuste ja Nginx Ingressiga. Migratsioon toimus kahes etapis - esmalt kirjutasime teenuste konfiguratsioonid ümber Podman Quadlet failideks, seejärel juurutasime need Kubernetese süsteemile DigitalOcean testimiskeskkonnas. Kaheetapiline lähenemine võimaldas valideerida iga muudatus eraldi ning vähendada riski, et kõik konfiguratsioonid tuleb korraga ümber kirjutada.

Suurimaks väljakutseks osutus Arete teenuste migreerimine Kubernetesesse. Arete teenused olid algselt arendatud eeldusega, et need töötavad Dockeris `network_mode: host` ja `privileged: true` režiimis, mis lihtsustas teenustevahelist suhtlust ja testijate konteinerite käivitamist, kuid muutis Kubernetese konfiguratsioonide koostamise oluliselt keerulisemaks. Teenuste käivitamiseks Kuberneteses tuli ümber kujundada võrgukonfiguratsioon, asendades hosti võrgurežiim Kubernetese sisevõrguga ning seadistades teenuste omavahelise suhtluse DNS-nimede kaudu. Lisaks nõudis teenuste sõltuvuste ahel hoolikat initsialiseerimisakonteinerite seadistamist. Nende muudatuste tegemine ja valideerimine võttis planeeritust oluliselt rohkem aega.

Koormustestimise tulemused kinnitasid süsteemi skaleeritavuse paranemist võrreldes varasema ühe serveri lahendusega. Varasemas lahenduses puudus päringute puhverdamine - koormuse tipphetkedel langes teenus täielikult ning üle 100 samaaegse kasutajaga muutus see kättesaamatuks. Uues Kubernetese lahenduses käitub süsteem põhimõtteliselt erinevalt: kuni 70 samaaegse kasutajaga võtab süsteem vastu 99,9% päringuid vigadeta. Kriitilise koormuse punkt saabub 100-150 kasutaja juures, kuid süsteem töötab edasi kontrollitud

jõudluse langusega - RabbitMQ puhverdab järjekorda ning ükski esitus ei lähe kaduma.

Automaatse taastumise test näitas, et podi rikke korral taastub süsteem automaatselt ligikaudu 25 sekundi jooksul ilma käsitsi sekkumiseta. See on põhimõtteline paranemine võrreldes varasema lahendusega, kus taastumine võiks võtta kuni 14 tundi. Klastri esmakordne seadistamine ja kõigi teenuste juurutamine võttis aega ligikaudu üks tund- see on märkimisväärne paranemine võrreldes varasema tulemüüri intsidendiga, kus teenuste taastamiseks kulus 12-14 tundi. Lisaks on deklaratiivne konfiguratsioon YAML-failidena ühtlasi dokumentatsioon - erinevalt varasemast olukorrast, kus süsteemi seadistus oli hajutatud ning puudus ühtne ülevaade infrastruktuurist.

Andmebaasihalduse osas seadistasime CloudNativePG operaatori kahe eksemplariga - esmane ja replika. Koormustestid kinnitasid andmebaasi töökindlust: kõigis testides oli vigade arv null ning replikatsiooni viivitus püsis 0 baiti, mis kinnitab CloudNativePG sünkroonse replikatsiooni töökindlust. RabbitMQ juurutati koos püsiva salvestusmahu taotlusega, mis lahendab varasema probleemi, kus sõnumijärjekorras olevad esitused kadusid podi taaskäivituse korral.

Tuvastati ka mitmeid piiranguid. Test Runner töötab endiselt privilegeeritud režiimis, kuid kasutab Docker-in-Docker mudelit, kus privilegeeritud juurdepääs on isoleeritud podi tasandil -Test Runner ei oma ligipääsu hosti failisüsteemile ega teistele klastri komponentidele. Suure sisendi testimisel avastati kolm järjestikust piirangut - Nginx Ingressi, RabbitMQ ja Jackson teegi limiidid - mille lahenduseks oleks Garage objektisalvestuse integreerimine, kuid selleni käesoleva töö raames ei jõutud. Üksiku Test Runneri läbilaskevõime (0,4 päringut sekundis) ei ole piisav suurema samaaegse kasutajate arvu jaoks, kuid horisontaalne skaleerimine lahendaks selle probleemi tootmiskeskkonnas.

5.3 Edasiarendus

Bakalaureusetöö raames loodud seire- ja orkestreerimissüsteemi on tulevikus võimalik veelgi edasi arendada ja optimeerida.

Praegu salvestatakse jäljed Grafana Loki andmebaasi tavaliste logidena. Siiski sisaldab Grafana Stack eraldi jälgede haldamiseks mõeldud teenust nimega Grafana Tempo, mis onloodud hajusjälgimise andmete talletamiseks ja analüüsimiseks. Tulevikus oleks otstarbe-

kas viia jälgede salvestamine üle Tempole, et kasutada täielikult ära selle funktsionaalsust ja parandada jälgede analüüsimise võimalusi [21].

Bakalaureusetöö käigus ei jõutud Moodle'i pistikprogrammi logide saatmise sagedust optimeerida väiksemaks kui üks minut. Praegune lahendus jääb seetõttu ajaliselt piiratumaks võrreldes teiste teenustega, kus logid edastatakse märksa tihedamini. Tulevikus oleks soovitav sellele probleemile leida parem lahendus.

Lisaks oleks soovitatav põhjalikumalt integreerida Prometheus või Grafana häireteavituste süsteem, mis võimaldaks automaatselt teavitada arendajaid kriitiliste sündmuste korral, näiteks serveri töö katkemisel või oluliste süsteeminäitajate ületamisel.

Kubernetese platvormile migreerimise käigus tuvastati mitmeid valdkondi, mida on võimalik tulevikus parandada ja laiendada.

Käesolev töö keskendus Kubernetese konfiguratsioonide arendamisele ja valideerimisele DigitalOcean testimiskeskkonnas. Järgmise sammuna on plaanis Kubernetese klastri üleviimine tootmiskeskkonda, mis võimaldab rakendada kõiki käesolevas töös kirjeldatud lahendusi - automaatset taastumist, püsisalvestust ja deklaratiivset konfiguratsioonihaldust - reaalses tootmiskeskkonnas. Sellega koos on plaanis seadistada GitLabi Kubernetese agendi integratsioon, mis võimaldab koodimuudatustel käivitada automaatselt uue versiooni juurutuse klastris. GitLabi agenti ei kasutatud testimisfaasis meie GitLabi versiooni ja Kubernetese ühilduvusprobleemide tõttu, kuid tootmiskeskkonna seadistamisel on see prioriteetne samm.

Käesoleva töö raames töötab Test Runner privilegeeritud režiimis (`privileged: true`), kasutades Docker-in-Docker mudelit, kus põhikonteiner suhtleb kõrvalkonteineriga (*sidecar*) läbi sisevõrgu liidese `DOCKER_HOST=tcp://localhost:2375`. See tähendab, et privilegeeritud juurdepääs on isoleeritud podi tasandil - Test Runner ei oma ligipääsu hosti failisüsteemile ega teistele klastri komponentidele, vaid suhtleb ainult Docker-in-Docker kõrvalkonteineriga. Sellest hoolimata on privilegeeritud režiim Kubernetese keskkonnas ebasoovitav turvapoliitika seisukohalt ning vajab edasist lahendust. Tulevikus on plaanis asendada praegune lahendus Kubernetes Job API-põhise lähenemisega, kus iga koodiesituse jaoks luuakse eraldi Kubernetese töö (*job*), mis käivitab testija konteineri isoleeritud keskkonnas. See lähenemine kõrvaldab vajaduse privilegeeritud režiimi järele, kuna testijate

konteinerite halduse võtab üle Kubernetes ise, mitte Docker-in-Docker. Selline muudatus nõuab Test Runneri koodibaasi ümberkirjutamist.

Koormustestimise käigus selgus, et üksik Test Runner eksemplar ei suuda teenindada 70 või enamat samaaegset kasutajat - läbilaskevõime jäi ligikaudu 0,4 päringut sekundis. Test Runneri horisontaalne skaleerimine mitme replikateni lahendaks selle kitsaskoha, kuid seda ei olnud võimalik DigitalOcean testimiskeskkonnas täielikult valideerida, kuna klaster piirdus kolme sõlmega. Oma tootmisklastris on replikate arvu suurendamine lihtne konfiguratsiooliline muudatus - tulevikus on plaanis seadistada horisontaalne automaatne skaleerimine (ingl. k *Horizontal Pod Autoscaler*, HPA) RabbitMQ järjekorra pikkuse põhjal, mis võimaldab süsteemil automaatselt kohandada Test Runneri replikate arvu vastavalt hetkekoormusele.

Käesoleva töö raames puudusid mehhanismid suuremahuliseks koormustestimiseks - testimine piirdus k6 tööriistaga kuni 150 samaaegse kasutajaga ning DigitalOcean testimiskeskkonnas oli klastri suurus piiratud. Tulevikus on plaanis viia läbi põhjalikum koormustestimine tootmisklastris, mis võimaldab testida süsteemi käitumist reaalse eksamisituatsiooni koormusega ning valideerida horisontaalse skaleerimise mõju läbilaskevõimele.

Nginx Ingressi kontrolleril põhineva lahenduse asendamine Gateway API-ga on samuti plaanis. Kubernetes SIG Network poolt hallatav Ingress NGINX Controller lõpetas aktiivselt arendamise 2026. aasta märtsis ning ei saa enam turvaparandusi ega veaparandusi [53]. Gateway API on Kubernetese kogukonna poolt soovitatud kaasaegne lahendus, mis pakub paindlikumat ja võimsamat liikluse marsruutimist [54]. Nginx Ingressi kasutamine oli praeguses etapis põhjendatud, kuna see võimaldas migreerimist lihtsustada ilma täiendava konfiguratsioonikeerukuseta.

Käesolevas töös valiti objektisalvestuse lahenduseks Garage, kuid selle integreerimine teenustega jäi käesoleva töö skoobist välja. Nagu peatükis 2.1 kirjeldatud, põhjustab praegune olukord kaks probleemi: esiteks lähevad serveri rikke korral lokaalsesse failisüsteemi salvestatud failid kaotsi, teiseks ületavad suuremad failid RabbitMQ sõnumi suuruse limiiti 128 MB. Garage integreerimine lahendaks mõlemad probleemid korraga - failid salvestataks S3 protokolli kaudu Garage'isse ning RabbitMQ kaudu edastataks ainult viide salvestatud failile. Garage seadistamine kahe replikaga tagaks, et iga fail on salvestatud kahes kohas

ning ühe sõlme rike ei tooks kaasa andmekadu. Integreerimine eeldab teenuste koodibaasi muutmist, mistõttu jääb see edasise arendustöö osaks.

Lisaks on plaanis Arete teenuste ümberkirjutamine Kotlin programmeerimiskeelde, mis parandab teenuste hooldatavust ja jõudlust. Ümberkirjutamise käigus on võimalik rakendada käesolevas töös kirjeldatud logimisstandard ja mõõdikute kogumine algusest peale. Eelkõige on võimalik teenused arendada klastrit arvestades - kasutades raamistikke, mis võimaldavad teenustel suhelda Kubernetese klastriga otse, jagada andmeid teenuste vahel ning kasutada klastri funktsioone täiel määral. See on oluline samm edasi praegusest lahendusest, kus Kubernetese tugi on lisatud olemasolevate teenuste peale, mitte ehitatud sisse algusest peale.

Süsteemi põhjalikum testimine ja valideerimine on samuti plaanis - sealhulgas Test Runneri horisontaalse skaleerimise testimine reaalse eksamisituatsiooni koormusega, GitLabi Kubernetese agendi integratsioon ning teenuste jõudluse mõõtmine tootmiskeskkonnas.

6 Kokkuvõte

Käesoleva bakalaureusetöö eesmärk oli lahendada Kooud ja Arete tarkvarasüsteemi peamised operatsioonilised kitsaskohad kolme alaeesmärgi kaudu: ühtlustada logimine kõikides teenustes, luua keskne seire- ja analüüsilahendus ning migreerida teenused Kubernetese süsteemile.

Logimise ühtlustamiseks töötasime välja ühtse logimisstruktuuri ning rakendasime selle Kooud tagarakenduses, *Request Forwarderis* ja *Test Runneris*. Keskse seirelahendusena juurutasime Grafana Stacki, mis koondab kõigi teenuste logid ja mõõdikud ühte kohta ning võimaldab arendajatel analüüsida süsteemi seisundit ilma käsitsi iga teenuse logisid kontrollimata. Loodud logimislahenduse praktiline väärtus leidis kinnitust juba arendus- ja testimisfaasis - Grafana logid aitasid tuvastada vea Moodle'i hinnete sünkroniseerimisel, mis varasemalt oleks jäänud märkamatuks.

Kubernetese süsteemile migreerimine toimus kahes etapis - esmalt kirjutasime teenuste konfiguratsioonid ümber Podman Quadlet failideks, seejärel juurutasime teenused Kubernetese platvormile DigitalOcean testimiskeskkonnas. Rakendasime RabbitMQ püsisalvestuse podi taaskäivituste üle, automaatse taastumise *self-healing* mehhanismi ning CloudNativePG andmebaasiklastri kahe eksemplariga, mis tagab automaatse tõrkeülekandumise andmebaasi rikke korral. Koormustestimine kinnitas, et kuni 70 samaaegse kasutajaga võtab süsteem vastu 99,9% päringuid vigadeta ning podi rikke korral taastub süsteem automaatselt ligikaudu 25 sekundi jooksul - varasemas lahenduses võttis sama protsess kuni 14 tundi. Kriitilise koormuse punkt saabub 100-150 samaaegse kasutaja juures, kuid süsteem töötab edasi kontrollitud jõudluse langusega - RabbitMQ puhverdab järjekorda ning ükski esitus ei lähe kaduma.

Loodud lahendus loob hea aluse edasisteks arendusteks. Lähitulevikus on plaanis viia Kubernetese klaster tootmiskeskkonda ning seadistada GitLabi Kubernetese agendi integratsioon automaatseks juurutamiseks. Pikemas perspektiivis on kaalutud Arete teenuste

ümberkirjutamist Kotlin programmeerimiskeelde, mis võimaldaks teenused arendada klastrit arvestades ning rakendada logimisstandard algusest peale.

Kasutatud kirjandus

[1] Joosep Franz Moorits Alviste. *Charon's bachelor's theses*. Kasutatud: 03-05-2026. URL: https://digikogu.taltech.ee/et/item/b628d504-57e3-4d90-9c60-4bcd6bea3d61.

[2] Enrico Vompa. *Arete's bachelor's theses*. Kasutatud: 03-05-2026. URL: https://digikogu.taltech.ee/et/Item/11f7fa57-a742-4449-bdba-b4a700e4db76.

[3] Chrystal R. China. *What is observability?* [Kasutatud: 25-04-2026]. 2024. URL: https://www.ibm.com/think/topics/observability.

[4] Chrystal R. China. *Observability versus monitoring: What's the difference?* [Kasutatud: 17-05-2026]. 2025. URL: https://www.ibm.com/think/topics/observability-vs-monitoring.

[5] R. Gerhards. *The Syslog Protocol*. RFC 5424 (Standards Track). [Kasutatud: 26-04-2026]. 2009. URL: https://datatracker.ietf.org/doc/rfc5424.

[6] Kubernetes. *Overview*. Official documentation. [Kasutatud: 14-03-2026]. URL: https://kubernetes.io/docs/concepts/overview/.

[7] Cloud Native Computing Foundation. *CNCF — Cloud Native Computing Foundation*. Official website. [Kasutatud: 14-03-2026]. URL: https://www.cncf.io/.

[8] The Kubernetes Authors. *Kubernetes Documentation*. Official documentation. [Kasutatud: 25-04-2026]. URL: https://kubernetes.io/docs/home/.

[9] The Kubernetes Authors. *Kubernetes Components*. Official documentation. [Kasutatud: 23-04-2026]. URL: https://kubernetes.io/docs/concepts/overview/components/.

[10] Kubernetes. *Namespaces*. Official documentation. [Kasutatud: 21-03-2026]. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[11] Kubernetes. *Pods*. Official documentation. [Kasutatud: 21-03-2026]. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[12] Kubernetes. *Deployments*. Official documentation. [Kasutatud: 21-03-2026]. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[13] Kubernetes. *Service*. Official documentation. [Kasutatud: 28-03-2026]. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[14] Kubernetes. *ConfigMaps*. Official documentation. [Kasutatud: 28-03-2026]. URL: https://kubernetes.io/docs/concepts/configuration/configmap/.

[15] Kubernetes. *Secrets*. Official documentation. [Kasutatud: 28-03-2026]. URL: https://kubernetes.io/docs/concepts/configuration/secret/.

[16] Kubernetes. *Persistent Volumes*. Official documentation. [Kasutatud: 04-04-2026]. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[17] Kubernetes. *Ingress*. Official documentation. [Kasutatud: 04-04-2026]. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/.

[18] Helm. *Helm Documentation*. Official documentation. [Kasutatud: 04-04-2026]. URL: https://helm.sh/docs/.

[19] CloudNativePG. *CloudNativePG Documentation*. Official documentation. [Kasutatud: 11-04-2026]. URL: https://cloudnative-pg.io/documentation/.

[20] Elastic. *What is the ELK Stack?* Official documentation. [Kasutatud: 26-04-2026]. URL: https://www.elastic.co/what-is/elk-stack.

[21] Grafana Labs. *The Grafana Stack*. Official Documentation. [Kasutatud: 26-04-2026]. URL: https://grafana.com/docs/.

[22] AWS. *Announcing Amazon OpenSearch Service which Supports OpenSearch 1.0*. Announce. [Kasutatud: 27-04-2026]. URL: https://aws.amazon.com/blogs/aws/announcing-amazon-opensearch-service-which-supports-opensearch-10/.

[23] Elastic. *Logstash Reference Documentation*. Official documentation. [Kasutatud: 26-04-2026]. URL: https://www.elastic.co/guide/en/logstash/current/index.html.

[24] Elastic. *Elacticsearch license*. [Kasutatud: 26-04-2026]. URL: https://www.elastic.co/pricing/faq/licensing.

[25] Trail of Bits. *OpenSearch*. Benchmark assesment. [Kasutatud: 27-04-2026]. URL: https://github.com/trailofbits/publications/blob/master/reports/OpenSearch-Benchmarking.pdf.

[26] Grafana Labs. *Grafana Documentation*. Official Documentation. [Kasutatud: 26-04-2026]. URL: https://grafana.com/docs/grafana/latest/.

[27] Grafana Labs. *Grafana Loki Documentation*. Official Documentation. [Kasutatud: 26-04-2026]. URL: https://grafana.com/docs/loki/latest/.

[28] Grafana Labs. *Mimir documentation*. Official documentation. [Kasutatud: 27-04-2026]. URL: https://grafana.com/docs/mimir/latest/introduction/.

[29] Prometheus. *Prometheus documentation*. Official documentation. [Kasutatud: 27-04-2026]. URL: https://prometheus.io/docs/introduction/overview/.

[30] Docker. *Docker Swarm*. Official documentation. [Kasutatud: 18-04-2026]. URL: https://docs.docker.com/engine/swarm/.

[31] HashiCorp. *Terraform Documentation*. Official documentation. [Kasutatud: 16-05-2026]. URL: https://developer.hashicorp.com/terraform/docs.

[32] HashiCorp. *Nomad Documentation*. Official documentation. [Kasutatud: 18-04-2026]. URL: https://developer.hashicorp.com/nomad/docs.

[33] GitLab. *GitLab Kubernetes Agent*. Official documentation. [Kasutatud: 02-05-2026]. URL: https://docs.gitlab.com/ee/user/clusters/agent/.

[34] Amazon Web Services. *Amazon S3*. [Kasutatud: 02-05-2026]. URL: https://aws.amazon.com/s3/.

[35] MinIO. *MinIO License*. Official documentation. [Kasutatud: 16-05-2026]. URL: https://min.io/pricing.

[36] Garage. *Garage Documentation*. Official documentation. [Kasutatud: 02-05-2026]. URL: https://garagehq.deuxfleurs.fr/documentation/.

[37] Internet Security Research Group (ISRG). *Let's Encrypt*. Kasutatud: 18-05-2026. URL: https://letsencrypt.org/.

[38] Electronic Frontier Foundation (EFF). *Certbot*. Kasutatud: 18-05-2026. URL: https://certbot.eff.org/.

[39] Documentation. *prometheus-net documentation*. Official documentation. [Kasutatud: 11-05-2026]. URL: https://github.com/prometheus-net/prometheus-net.

[40] Documentation. *node exporter documentation*. Official documentation. [Kasutatud: 11-05-2026]. URL: https://github.com/prometheus/node_exporter.

[41] Podman. *Podman Documentation*. Official documentation. [Kasutatud: 25-04-2026]. URL: https://docs.podman.io/.

[42] Kubernetes SIGs. *kind - Kubernetes IN Docker*. Official documentation. [Kasutatud: 25-04-2026]. URL: https://kind.sigs.k8s.io/.

[43] DigitalOcean. *DigitalOcean Kubernetes Service*. Official documentation. [Kasutatud: 30-05-2026]. URL: https://docs.digitalocean.com/products/kubernetes/.

[44] Nginx. *Nginx Ingress Controller Documentation*. Official documentation. [Kasutatud: 09-05-2026]. URL: https://docs.nginx.com/nginx-ingress-controller/.

[45] Podman. *Quadlet*. Official documentation. [Kasutatud: 25-04-2026]. URL: https://docs.podman.io/en/latest/markdown/podman-systemd.unit.5.html.

[46] RabbitMQ. *RabbitMQ Documentation*. Official documentation. [Kasutatud: 18-04-2026]. URL: https://www.rabbitmq.com/docs.

[47] Spring. *Spring Boot Actuator*. Official documentation. [Kasutatud: 09-05-2026]. URL: https://docs.spring.io/spring-boot/reference/actuator/index.html.

[48] Grafana Labs. *k6 Load Testing*. Official documentation. [Kasutatud: 30-05-2026]. URL: https://k6.io/.

[49] FasterXML. *Jackson JSON processor*. [Kasutatud: 31-05-2026]. URL: https://github.com/FasterXML/jackson.

[50] PostgreSQL. *pgbench*. [Kasutatud: 31-05-2026]. URL: https://www.postgresql.org/docs/current/pgbench.html.

[51] Transaction Processing Performance Council. *TPC-B Benchmark*. [Kasutatud: 31-05-2026]. URL: https://www.tpc.org/tpcb/.

[52] PostgreSQL. *Explicit Locking*. [Kasutatud: 31-05-2026]. URL: https://www.postgresql.org/docs/current/explicit-locking.html.

[53] Kubernetes. *Ingress NGINX Controller Deprecation*. [Kasutatud: 19-05-2026]. URL: https://opensource.googleblog.com/2026/02/the-end-of-an-era-transitioning-away-from-ingress-nginx.html.

[54] Kubernetes. *Gateway API*. [Kasutatud: 19-05-2026]. URL: https://gateway-api.sigs.k8s.io/.

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

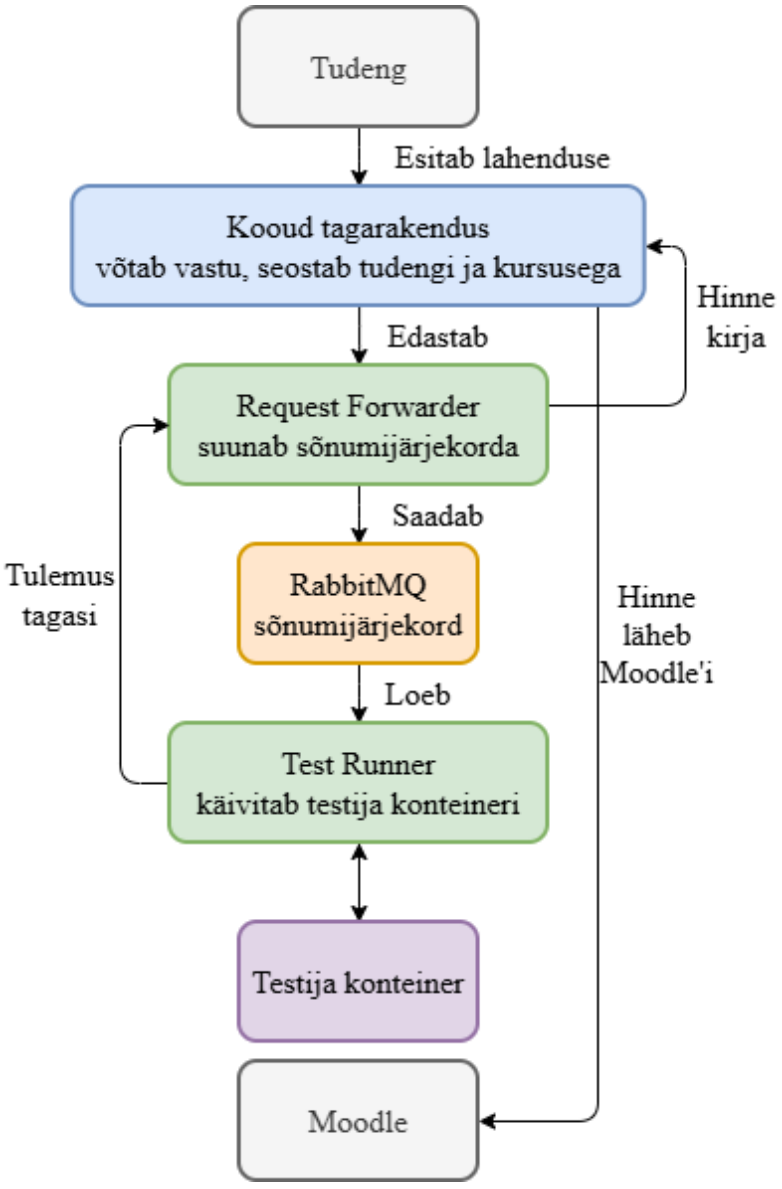
Meie, Anton Budovey ja Artjom Hruljov

1. Anname Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Kooud ja Arete teenuste migreerimine skaleeruvale arhitektuurile seire toega", mille juhendaja on Bahdan Yanovich
    1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
    1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Oleme teadlikud, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autoritele.
3. Kinnitame, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

12.06.2026

[1] Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

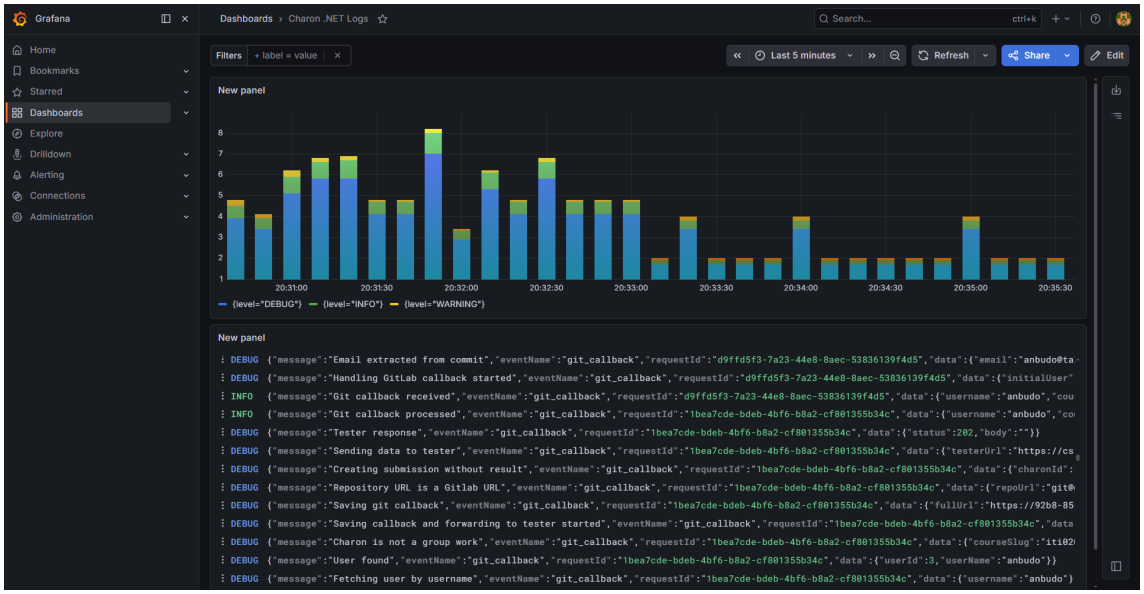
Lisa 2 – Programmeerimisülesannete testimisvoog



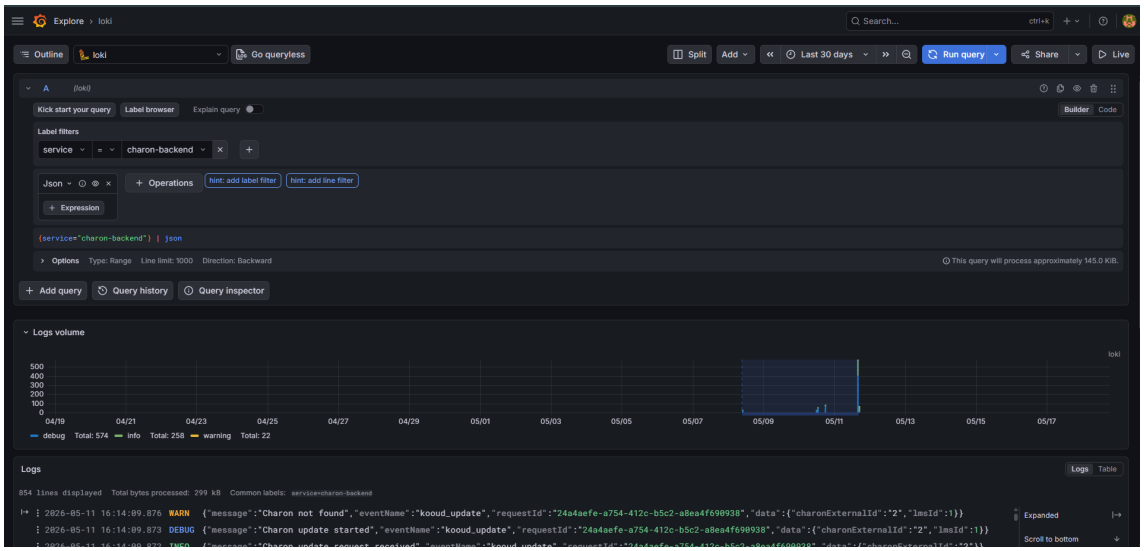
Lisa 3 – Loki logide saatmise JSON-keha näide

```
{
  "streams": [
    {
      "stream": {
        "service": "kooud-backend",
        "level": "INFO"
      },
      "values": [
        [
          "1777305364699000000",
          "{
            "message": "Kooud update request received",
            "event_name": "kooud-update",
            "request_id": "97490650-e590-43e3-8a79-70ecf8c5557d",
            "data": {
              "charonId": 4,
              "courseId": 2,
              "lmsId": 1
            }
          }"
        ]
      ]
    }
  ]
}
```

Lisa 4 – Uue logimisstruktuuri visualiseerimine Grafana keskkonnas



Lisa 5 – Logide filtreerimine märgiste abil Grafana keskkonnas



Lisa 6 – Nimeruumi konfiguratsioon

Joonis 6.1. namespace.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: arete
  labels:
    project: arete
    managed-by: kubectl
```

Lisa 7 – Podman Quadlet failid

Joonis 7.1. arete-charon.container

```
[Unit]
Description=Charon Backend Service
After=network-online.target
Requires=arete-postgres.service
After=arete-postgres.service

[Container]
Image=gitlab.cs.taltech.ee:5050/kooud/arete/charon-backend:latest
ContainerName=arete-charon
Network=arete.network
EnvironmentFile=/etc/arete/.env
Environment=ASPNETCORE_URLS=http://0.0.0.0:5000
Environment=ENCRYPTOR_BASE64=<REDACTED>
Environment=ConnectionStrings__DefaultConnection=Host=arete-
   postgres;
  Port=5432;Database=charon;Username=<REDACTED>;Password=<REDACTED>
PublishPort=5000:5000
Label=service=charon
Label=project=arete

[Service]
Restart=always
TimeoutStartSec=60

[Install]
WantedBy=multi-user.target
```

Joonis 7.2. arete-postgres.container

```
[Unit]
Description=Arete PostgreSQL Database
After=network-online.target
```

```
[Container]
Image=docker.io/postgres:15
ContainerName=arete-postgres
Network=arete.network
EnvironmentFile=/etc/arete/.env
Environment=POSTGRES_USER=<REDACTED>
Environment=POSTGRES_PASSWORD=<REDACTED>
Environment=POSTGRES_DB=arete
Volume=arete-postgres.volume:/var/lib/postgresql/data
PublishPort=5432:5432
Label=service=postgres
Label=project=arete

[Service]
Restart=always
TimeoutStartSec=120

[Install]
WantedBy=multi-user.target
```

Joonis 7.3. arete-rabbitmq.container

```
[Unit]
Description=Arete RabbitMQ Message Broker
After=network-online.target

[Container]
Image=docker.io/rabbitmq:3-management
ContainerName=arete-rabbitmq
Network=arete.network
EnvironmentFile=/etc/arete/.env
Environment=RABBITMQ_DEFAULT_USER=<REDACTED>
Environment=RABBITMQ_DEFAULT_PASS=<REDACTED>
Environment=RABBITMQ_DEFAULT_VHOST=/test-runner
Volume=/etc/arete/rabbitmq.conf:/etc/rabbitmq/conf.d/20-custom.conf
Volume=arete-rabbitmq.volume:/var/lib/rabbitmq
PublishPort=5672:5672
PublishPort=15672:15672
Label=service=rabbitmq
Label=project=arete
```

```
[Service]
Restart=always
TimeoutStartSec=60

[Install]
WantedBy=multi-user.target
```

Joonis 7.4. arete-request-forwarder.container

```
[Unit]
Description=Arete Request Forwarder Service
After=network-online.target
Requires=arete-postgres.service
After=arete-postgres.service
Requires=arete-rabbitmq.service
After=arete-rabbitmq.service

[Container]
Image=gitlab.cs.taltech.ee:5050/kooud/arete/arete-request-forwarder
   :latest
ContainerName=arete-request-forwarder
Network=arete.network
EnvironmentFile=/etc/arete/.env
Environment=DATABASE_URI=arete-postgres
Environment=DATABASE_PORT=5432
Environment=DATABASE_PASSWORD=<REDACTED>
Environment=RABBITMQ_HOST=arete-rabbitmq
Environment=RABBITMQ_USER=<REDACTED>
Environment=RABBITMQ_PASSWORD=<REDACTED>
Environment=BACKEND_TOKEN=<REDACTED>
Environment=REGISTRY_HOST=<REDACTED>
Environment=GITLAB_REGISTRY_USER=<REDACTED>
Environment=GITLAB_REGISTRY_TOKEN=<REDACTED>
Environment=ARETE_HOME=${ARETE_HOME}
Environment=LOG_LEVEL=${LOG_LEVEL}
Volume=${ARETE_HOME}:${ARETE_HOME}:z
Volume=${ARETE_HOME}/ssh:/root/.ssh:ro,z
PublishPort=8098:8098
Label=service=request-forwarder
```

```
Label=project=arete

[Service]
Restart=always
TimeoutStartSec=120

[Install]
WantedBy=multi-user.target
```

Joonis 7.5. arete-test-runner.container

```
[Unit]
Description=Arete Test Runner Service
After=network-online.target
Requires=arete-rabbitmq.service
After=arete-rabbitmq.service

[Container]
Image=gitlab.cs.taltech.ee:5050/kooud/arete/arete-test-runner:
    latest
ContainerName=arete-test-runner
Network=arete.network
EnvironmentFile=/etc/arete/.env
Environment=RABBITMQ_HOST=arete-rabbitmq
Environment=RABBITMQ_USER=<REDACTED>
Environment=RABBITMQ_PASSWORD=<REDACTED>
Environment=REGISTRY_HOST=<REDACTED>
Environment=GITLAB_REGISTRY_USER=<REDACTED>
Environment=GITLAB_REGISTRY_TOKEN=<REDACTED>
Environment=TESTRUNNER_HOME=${TESTRUNNER_HOME}
Environment=NAME=testrunner
Environment=LOG_LEVEL=${LOG_LEVEL}
Volume=/var/run/docker.sock:/var/run/docker.sock
Volume=${TESTRUNNER_HOME}:${TESTRUNNER_HOME}
Volume=${ARETE_HOME}/ssh:/root/.ssh
PublishPort=8098:8098
PodmanArgs=--privileged
Label=service=test-runner
Label=project=arete
```

```
[Service]
Restart=always
TimeoutStartSec=120

[Install]
WantedBy=multi-user.target
```

Lisa 8 – RabbitMQ konfiguratsioonifailid

Joonis 8.1. rabbitmq-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
  namespace: arete
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      containers:
        - name: rabbitmq
          image: docker.io/rabbitmq:3-management
          ports:
            - containerPort: 5672
            - containerPort: 15672
          env:
            - name: RABBITMQ_DEFAULT_USER
              valueFrom:
                secretKeyRef:
                  name: rabbitmq-credentials
                  key: username
            - name: RABBITMQ_DEFAULT_PASS
              valueFrom:
                secretKeyRef:
                  name: rabbitmq-credentials
                  key: password
```

```
  - name: RABBITMQ_DEFAULT_VHOST
    value: "/test-runner"
  - name: RABBITMQ_NODE_PORT
    value: "5672"
  - name: RABBITMQ_MANAGEMENT_PORT
    value: "15672"
  - name: RABBITMQ_LOOPBACK_USERS
    value: "none"
  - name: RABBITMQ_USE_LONGNAME
    value: "false"
  - name: RABBITMQ_NODENAME
    value: "rabbit@localhost"
  - name: RABBITMQ_SERVER_ADDITIONAL_ERL_ARGS
    value: "-rabbit vm_memory_high_watermark 0.4
      -rabbit vm_memory_high_watermark_paging_ratio 0.75"
volumeMounts:
  - name: rabbitmq-data
    mountPath: /var/lib/rabbitmq
resources:
  requests:
    memory: "256Mi"
    cpu: "100m"
  limits:
    memory: "512Mi"
    cpu: "500m"
startupProbe:
  tcpSocket:
    port: 5672
  failureThreshold: 30
  periodSeconds: 10
livenessProbe:
  tcpSocket:
    port: 15672
  initialDelaySeconds: 60
  periodSeconds: 15
  timeoutSeconds: 5
  failureThreshold: 3
readinessProbe:
  tcpSocket:
```

```
            port: 15672
          initialDelaySeconds: 60
          periodSeconds: 10
          timeoutSeconds: 5
          failureThreshold: 6
      volumes:
        - name: rabbitmq-data
          persistentVolumeClaim:
            claimName: rabbitmq-pvc
```

Joonis 8.2. rabbitmq-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: rabbitmq-pvc
  namespace: arete
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Joonis 8.3. rabbitmq-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq-service
  namespace: arete
spec:
  selector:
    app: rabbitmq
  ports:
    - name: amqp
      port: 5672
      targetPort: 5672
    - name: management
      port: 15672
      targetPort: 15672
```

```
type: ClusterIP
```

Lisa 9 – Request Forwarderi konfiguratsioonifailid

Joonis 9.1. request-forwarder-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: request-forwarder
  namespace: arete
spec:
  replicas: 1
  selector:
    matchLabels:
      app: request-forwarder
  template:
    metadata:
      labels:
        app: request-forwarder
    spec:
      initContainers:
        - name: wait-for-db
          image: busybox
          command:
            - sh
            - -c
            - until nc -z postgres-cluster-rw 5432;
              do echo "waiting for db..."; sleep 2; done
        - name: wait-for-rabbitmq
          image: busybox
          command:
            - sh
            - -c
            - until nc -z rabbitmq-service 5672;
              do echo "waiting for rabbitmq..."; sleep 2; done
      containers:
        - name: request-forwarder
```

```
image: gitlab.cs.taltech.ee:5050/kooud/arete/arete-
   request-forwarder:latest
ports:
  - containerPort: 8098
env:
  - name: DATABASE_URI
    valueFrom:
      configMapKeyRef:
        name: arete-config
        key: DATABASE_URI
  - name: DATABASE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: postgres-credentials
        key: password
  - name: RABBITMQ_HOST
    valueFrom:
      configMapKeyRef:
        name: arete-config
        key: RABBITMQ_HOST
  - name: RABBITMQ_USER
    valueFrom:
      secretKeyRef:
        name: rabbitmq-credentials
        key: username
  - name: RABBITMQ_PASSWORD
    valueFrom:
      secretKeyRef:
        name: rabbitmq-credentials
        key: password
  - name: LOG_LEVEL
    valueFrom:
      configMapKeyRef:
        name: arete-config
        key: LOG_LEVEL
  - name: LOKI_URL
    valueFrom:
      configMapKeyRef:
        name: arete-config
```

```
            key: LOKI_URL
    volumeMounts:
      - name: docker-socket
        mountPath: /var/run/docker.sock
      - name: arete-storage
        mountPath: /arete
      - name: ssh-keys
        mountPath: /root/.ssh
        readOnly: true
    resources:
      requests:
        memory: "512Mi"
        cpu: "250m"
      limits:
        memory: "1Gi"
        cpu: "1000m"
    livenessProbe:
      httpGet:
        path: /services/arete/api/v2/health
        port: 8098
      initialDelaySeconds: 60
      periodSeconds: 15
    readinessProbe:
      httpGet:
        path: /services/arete/api/v2/health
        port: 8098
      initialDelaySeconds: 45
      periodSeconds: 10
    imagePullPolicy: Always
    securityContext:
      privileged: true
volumes:
  - name: docker-socket
    hostPath:
      path: /var/run/docker.sock
      type: Socket
  - name: arete-storage
    persistentVolumeClaim:
      claimName: arete-storage-pvc
```

```
        - name: ssh-keys
          secret:
            secretName: arete-ssh-keys
            defaultMode: 0600
      imagePullSecrets:
        - name: gitlab-registry-credentials
```

Joonis 9.2. request-forwarder-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: request-forwarder-service
  namespace: arete
spec:
  selector:
    app: request-forwarder
  ports:
    - name: http
      port: 8098
      targetPort: 8098
  type: ClusterIP
```

Joonis 9.3. arete-storage-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: arete-storage-pvc
  namespace: arete
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Lisa 10 – Test Runneri konfiguratsioonifailid

Joonis 10.1. test-runner-deployment.yaml

```
apiVersion: apps/v1
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-runner
  namespace: arete
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-runner
  template:
    metadata:
      labels:
        app: test-runner
    spec:
      initContainers:
        - name: wait-for-rabbitmq
          image: busybox
          command:
            - sh
            - -c
            - until nc -z rabbitmq-service 5672;
              do echo "waiting for rabbitmq..."; sleep 2; done
      containers:
        - name: test-runner
          image: docker.io/daslon/arete-test-runner:latest
          ports:
            - containerPort: 8099
          env:
            - name: DOCKER_HOST
              value: "tcp://localhost:2375"
```

```
- name: RABBITMQ_HOST
  valueFrom:
    configMapKeyRef:
      name: arete-config
      key: RABBITMQ_HOST
- name: RABBITMQ_USER
  valueFrom:
    secretKeyRef:
      name: rabbitmq-credentials
      key: username
- name: RABBITMQ_PASSWORD
  valueFrom:
    secretKeyRef:
      name: rabbitmq-credentials
      key: password
- name: REGISTRY_HOST
  valueFrom:
    secretKeyRef:
      name: gitlab-registry-credentials
      key: registry-host
- name: GITLAB_REGISTRY_USER
  valueFrom:
    secretKeyRef:
      name: gitlab-registry-credentials
      key: registry-user
- name: GITLAB_REGISTRY_TOKEN
  valueFrom:
    secretKeyRef:
      name: gitlab-registry-credentials
      key: registry-token
- name: TESTRUNNER_HOME
  valueFrom:
    configMapKeyRef:
      name: arete-config
      key: TESTRUNNER_HOME
- name: LOG_LEVEL
  valueFrom:
    configMapKeyRef:
      name: arete-config
```

```
            key: LOG_LEVEL
      - name: NAME
        value: "testrunner"
      - name: LOKI_URL
        valueFrom:
          configMapKeyRef:
            name: arete-config
            key: LOKI_URL
    volumeMounts:
      - name: testrunner-storage
        mountPath: /testrunner
      - name: ssh-keys
        mountPath: /root/.ssh
        readOnly: true
      - name: machine-id
        mountPath: /etc/machine-id
        readOnly: true
    resources:
      requests:
        memory: "512Mi"
        cpu: "250m"
      limits:
        memory: "1Gi"
        cpu: "1000m"
    livenessProbe:
      tcpSocket:
        port: 8099
      initialDelaySeconds: 60
      periodSeconds: 15
      failureThreshold: 3
    readinessProbe:
      tcpSocket:
        port: 8099
      initialDelaySeconds: 45
      periodSeconds: 10
    imagePullPolicy: Always

  - name: dind
    image: docker:dind
```

```
            securityContext:
              privileged: true
            env:
              - name: DOCKER_TLS_CERTDIR
                value: ""
            volumeMounts:
              - name: docker-storage
                mountPath: /var/lib/docker
              - name: testrunner-storage
                mountPath: /testrunner
            resources:
              requests:
                memory: "512Mi"
                cpu: "250m"
              limits:
                memory: "2Gi"
                cpu: "1000m"

        volumes:
          - name: testrunner-storage
            persistentVolumeClaim:
              claimName: testrunner-storage-pvc
          - name: ssh-keys
            secret:
              secretName: arete-ssh-keys
              defaultMode: 0600
          - name: docker-storage
            emptyDir: {}
          - name: machine-id
            hostPath:
              path: /etc/machine-id
              type: File
        imagePullSecrets:
          - name: dockerhub-pull-secret
```

Joonis 10.2. test-runner-service.yaml

```
apiVersion: v1
kind: Service
metadata:
```

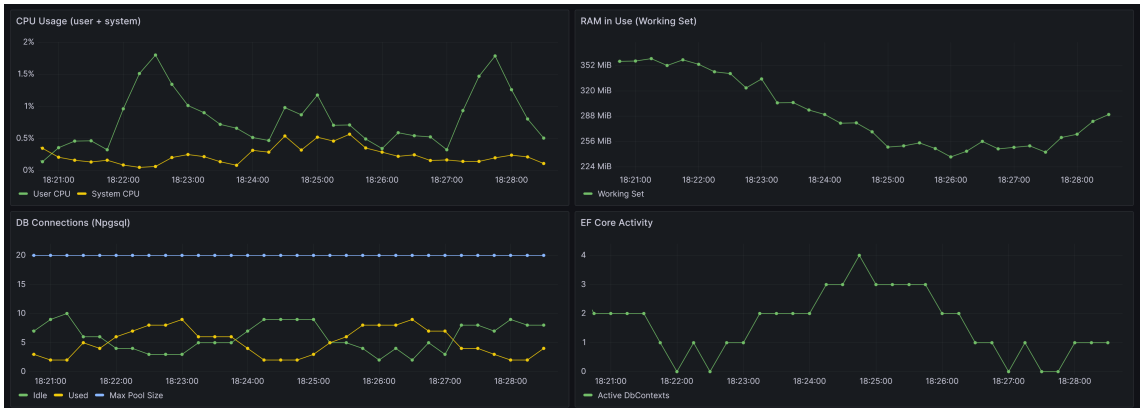
```
  name: test-runner-service
  namespace: arete
spec:
  selector:
    app: test-runner
  ports:
    - name: http
      port: 8099
      targetPort: 8099
  type: ClusterIP
```

Joonis 10.3. testrunner-storage-pvc.yaml

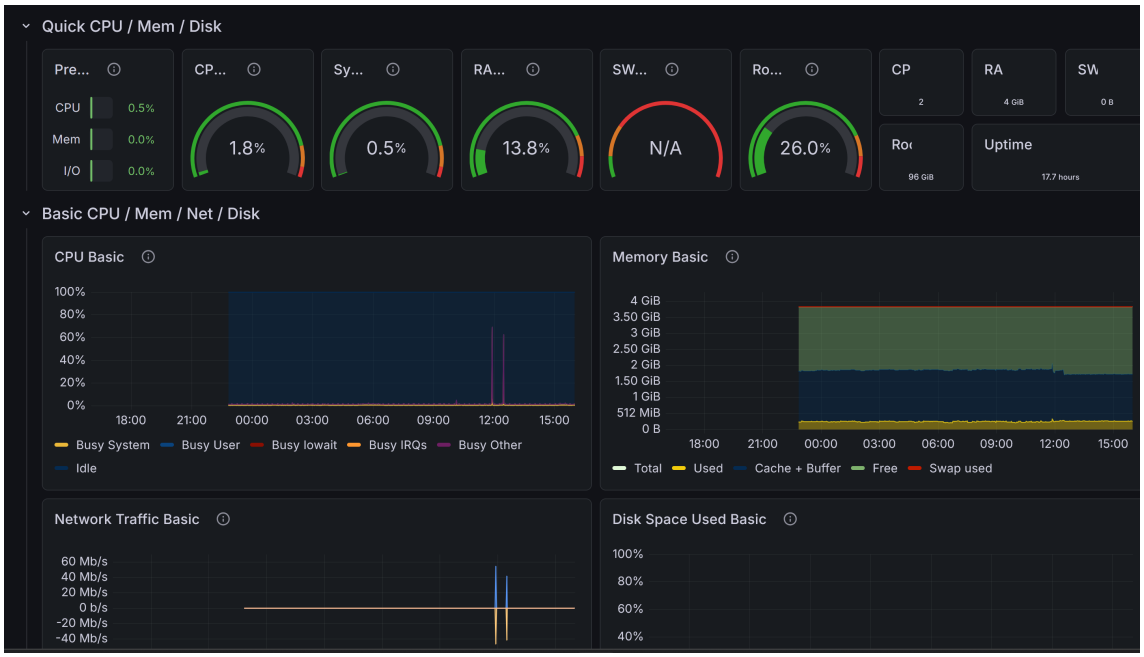
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: testrunner-storage-pvc
  namespace: arete
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Lisa 11 – Kooud esituste ning kursuste ja ülesannete aktiivsuse seiretöölaud

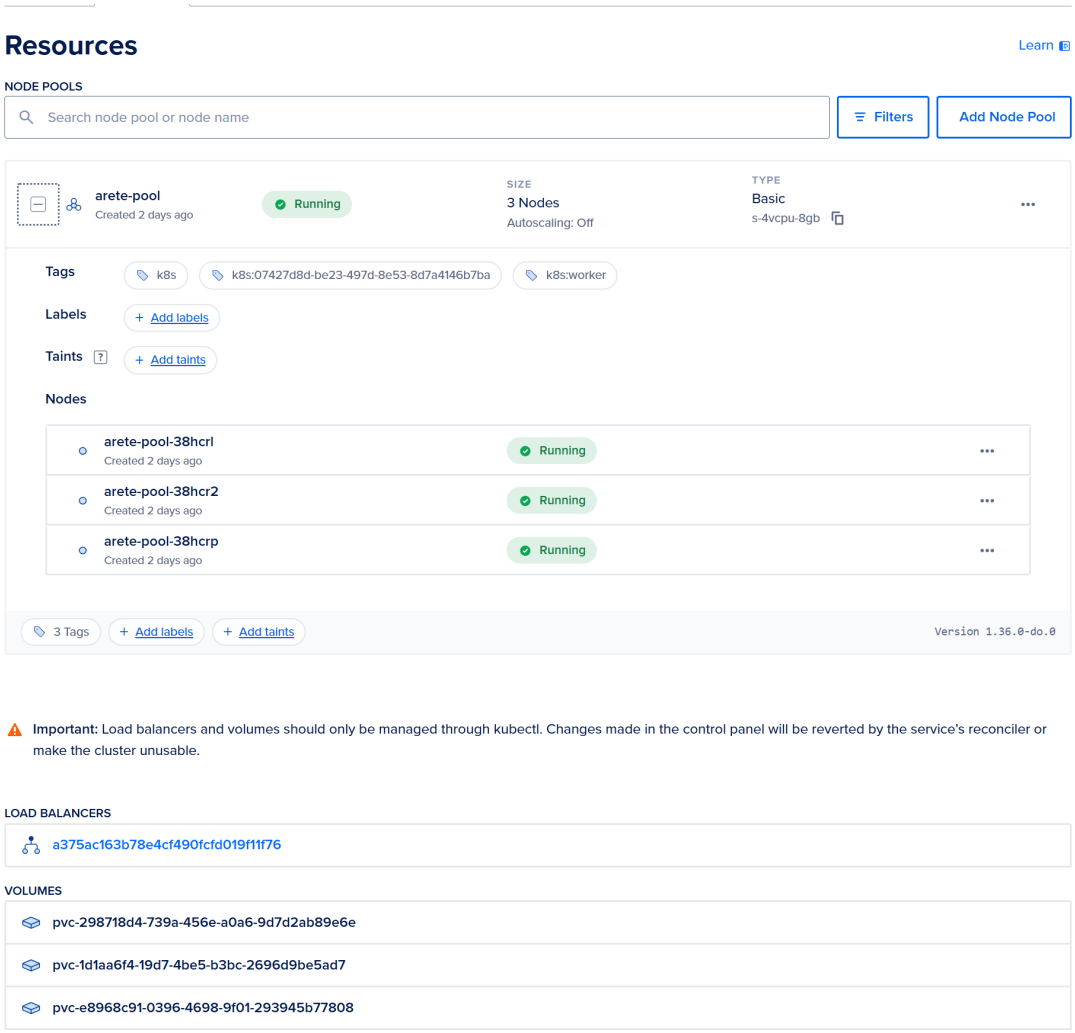
Lisa 12 – Kooud tagarakenduse süsteemimõõdikute ja ressursi-kasutuse töölaud



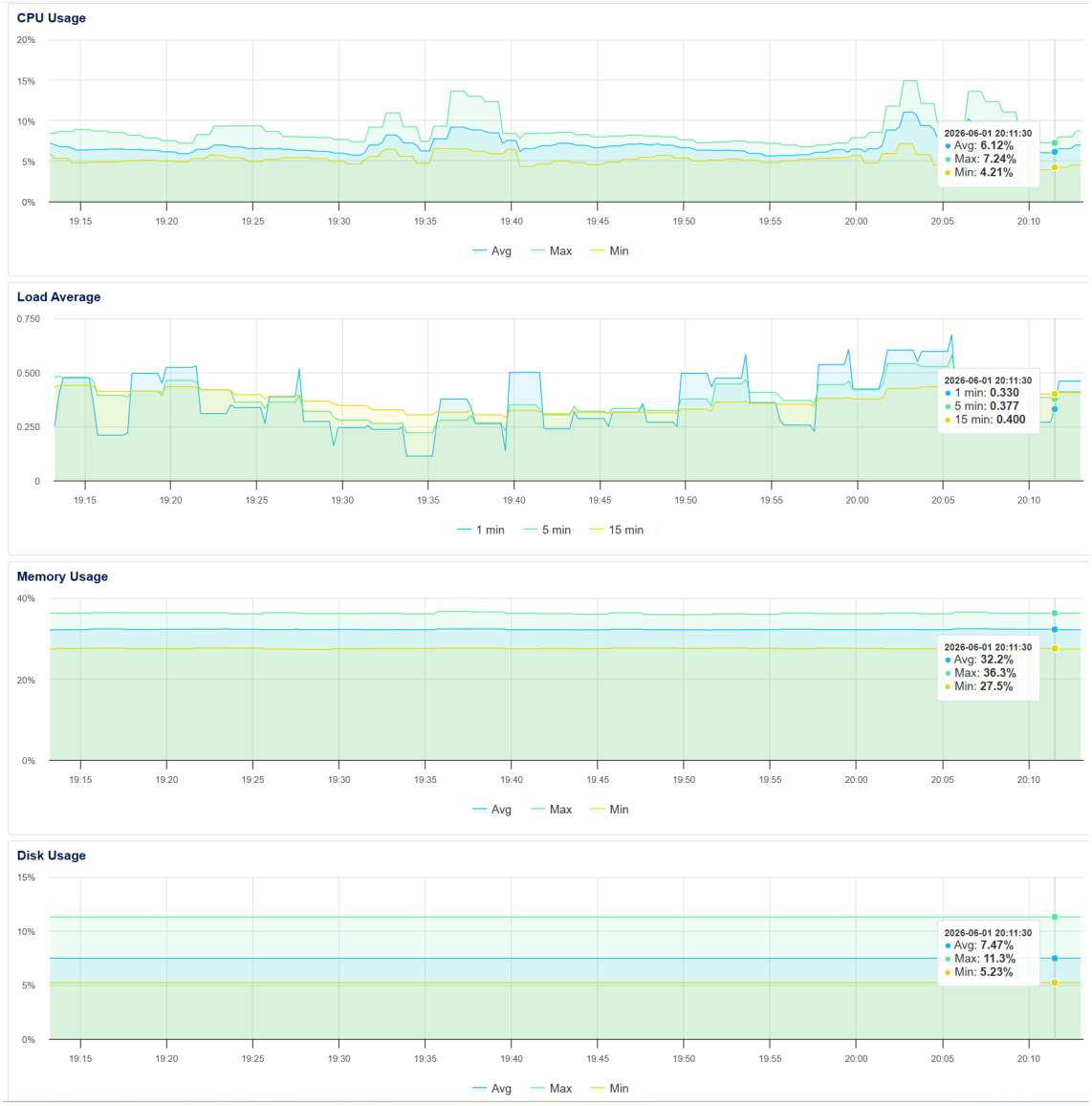
Lisa 13 – Serverite tehnilise seisundi töölaud (Node Exporter)



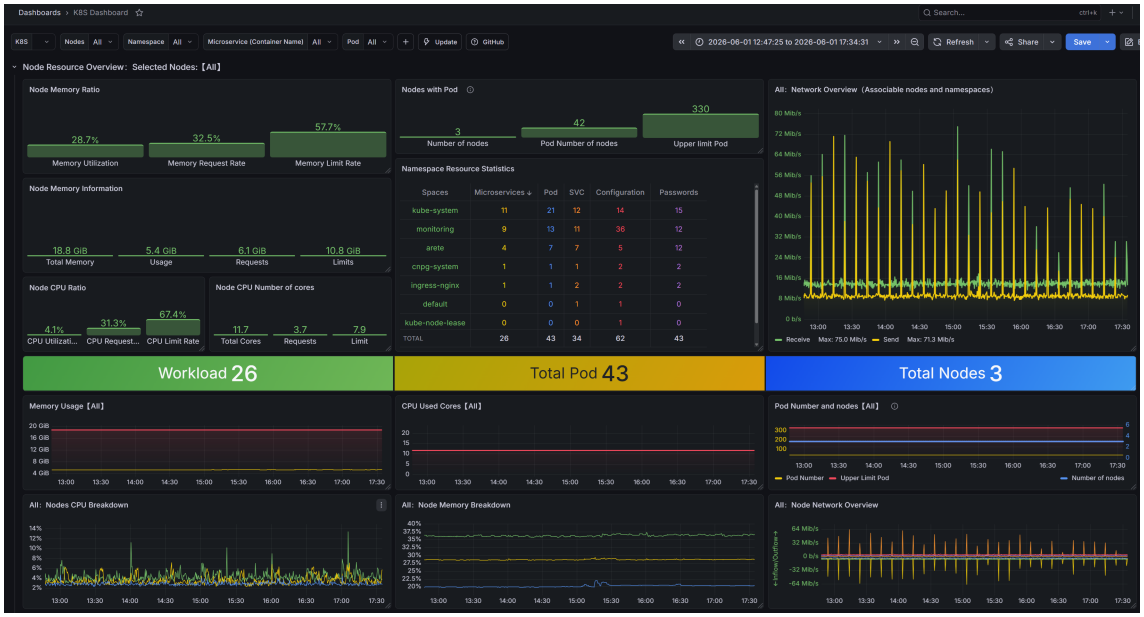
Lisa 14 – DigitalOcean klastri kolm töösõlme Running olekus



Lisa 15 – DigitalOcean klastri CPU, mälu ja ketta kasutus koormustestimise ajal



Lisa 16 – Kubernetese klastri ülevaate töölaud - nimeruumide statistika ja ressursikasutus



Lisa 17 – Kubernetese kolme töösõlme ressursikasutuse detail-vaade ja PVC salvestusmaht

Lisa 18 – Arete nimeruumi teenuste logid Grafana keskkonnas