TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aet-Kadi Kald 232883IAIB
Faina Dõmša 233693IAIB
Karoliina Kannik 233219IAIB

Programmeerimisülesannete haldamise registri Aurora ümberkirjutamine

Bakalaureusetöö

Juhendaja: Taavi Toimetaja
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: Aet-Kadi Kald, Faina Dõmša ja Karoliina Kannik

02.06.2026

Lühikokkuvõte

Käesoleva lõputöö eesmärk on programmeerimisülesannete registri Aurora ümberkirjutamine. Aurorat on arendatud mitme aasta jooksul, kuid süsteemis esineb endiselt palju vigu, mistõttu ei ole jõudnud see kunagi reaalsesse kasutusse. Rakenduse ümberkirjutamise vajadus tuleneb mitmest probleemist, näiteks aegunud tehnoloogiatest, turvanõrkustest ning mitmetest funktsionaalsustest, mis tegelikult ei tööta.

Teema on aktuaalne, kuna õppejõudude kasutatavate ülesannete hoidlate arv suureneb iga aastaga ning sobivate ülesannete leidmine eksamite ja kontrolltööde koostamiseks muutub järjest ajakulukamaks.

Töö käigus arendati uus esirakendus ja tagarakendus. Loodi uus kasutajaõiguste süsteem, parandati ülesannete kuvamist ja sarnasuse leidmist ning lisati uus funktsionaalsus ülesannete raskusastme hindamiseks tehisintellekti abil. Samuti lisati automaatne töövariantide koostamise süsteem.

Lõputöö tulemusena valmis rakendus, millel on olemas peamised funktsionaalsused ning dokumentatsioon, et muuta süsteemi edasine arendamine lihtsamaks ja arusaadavamaks. Lõputöö eesmärk saavutati, kuid töö käigus ilmnes ka oluline piirang: ülesannete korrektseks kuvamiseks peavad hoidlad järgima kindlat struktuuri.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 43 leheküljel, 6 peatükki, 4 joonist, 3 tabelit.

Abstract

Rewriting the Programming Assignment Management Registry Aurora

The aim of this thesis is to rewrite the programming assignment registry Aurora. Aurora has been developed over several years, but the system still contains many issues, which has prevented it from ever being used. The need to rewrite the application arose from several problems, including outdated technologies, security vulnerabilities, and multiple functionalities that do not work correctly.

The topic is relevant because the number of repositories used by lecturers increases every year, making the process of finding suitable assignments for exams and tests increasingly time-consuming.

During the development process, a new frontend and backend application were developed. A new user rights management system was created, assignment rendering and similarity detection were improved, and a new functionality for evaluating assignment difficulty using artificial intelligence was added. Additionally, an automatic exam compiler was created.

As a result of the thesis, a new application with the main core functionalities and documentation was completed to make further system development simpler and more understandable. The objective of the thesis was achieved; however, an important limitation emerged during development: repositories must follow a specific structure for assignments to be displayed correctly.

The thesis is in Estonian and contains 43 pages of text, 6 chapters, 4 figures, 3 tables.

Lühendite ja mõistete sõnastik

API	Rakendusliides (*Application Programming Interface*)
CRUD	Andmebaasi põhioperatsioonid (*Create, Read, Update, Delete*)
Entra ID	Microsofti pilvepõhine kasutajahaldusteenus
GitLab	Versioonihalduse platvorm
HTTP	Hüperteksti edastusprotokoll (*HyperText Transfer Protocol*)
IDE	Integreeritud programmeerimiskeskkond (*Integrated Development Environment*)
JWT	Kahe osapoole vaheline turvaline andmevahetusformaat (*JSON Web Token*)
LLM	Suur keelemudel (*Large Language Model*)
MAE	Keskmine absoluutne erinevus ennustatud ja tegeliku väärtuse vahel (*Mean Absolute Error*)
PAT	Isiklik juurdepääsuvõti (*Personal Access Token*)
PostgreSQL	Relatsiooniline andmebaasisüsteem
REST	Esitusoleku siire (*Representational State Transfer*)
SUS	Süsteemi kasutatavuse skaala (*System Usability Scale*)

Sisukord

Jooniste loetelu 9
Tabelite loetelu 10
1 Sissejuhatus 11
1.1 Metoodika 12
1.1.1 Tehisintellekt 13
2 Ülevaade probleemist ja eelnevast lahendusest 14
2.1 Sarnased lahendused 14
2.1.1 HackerRank 14
2.1.2 LeetCode 15
2.1.3 GitHub Classroom 16
2.2 Varasemad tööd 17
2.2.1 Süsteemi loomine 17
2.2.2 Õiguste ja statistika juurdearendus 17
2.2.3 Süsteemi stabiilsuse parendamine ja tehniline analüüs 19
2.2.4 Kasutajakogemuse parendamine 19
2.3 Järeldused varasematest töödest 20
2.4 Olemasoleva süsteemi analüüs 21
2.4.1 Tuvastatud toimivad funktsionaalsused 21
2.4.2 Tuvastatud probleemid 22
2.5 Uue lahenduse skoop 23
3 Loodava veebirakenduse ülevaade 24
3.1 Nõuete määramine 24
3.2 Tehnoloogiate valik 25
3.2.1 Tagarakendus 25
3.2.2 Esirakendus 26
3.3 Arhitektuur 26
3.4 Veebirakenduse disain 27

3.5 Põhifunktsionaalsuste loetelu ... 29
3.5.1 Autentimine Microsoftiga ... 29
3.5.2 GitLabi hoidlate tõmbamine ... 29
3.5.3 Õiguste ja gruppide süsteem ... 30
3.5.4 Ülesannete märkimine ... 32
3.5.5 Ülesannete sisu kuvamine ... 32
3.5.6 Ülesannete võrdlus ... 33
3.5.7 Sildistamissüsteem ... 34
3.5.8 Keerukuse hindamine ... 34
3.5.9 Töövariantide automaatne koostamine ... 34
4 Veebirakenduse arendus ... 36
4.1 Veebiteenuse-poolne lahendus ... 36
4.1.1 Autentimine Microsoftiga ... 36
4.1.2 Ülesannete keerukuse hindamine ... 37
4.1.3 Töövariantide automaatne koostamine ... 40
4.1.4 Sarnasus ... 41
4.1.5 Eksami ülesannete valimine ... 44
4.2 Testimine ja valideerimine ... 45
4.2.1 Testimine tellija esindajaga ... 46
4.2.2 Testimine arenduse käigus ... 46
4.2.3 Testimine Thymeleafi abil ... 47
4.2.4 Tiimisisene testimine ... 47
4.2.5 Valideerimine ... 48
5 Hinnang loodud veebirakendusele ... 50
5.1 Piirangud ... 50
5.2 Võimalused edasi arenduseks ... 51
6 Kokkuvõte ... 53
Kasutatud kirjandus ... 54
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks ... 57
Lisa 2 – Andmebaasi skeem ... 58
Lisa 3 – Ülesannete keerukuse hindamiseks kasutatud prompt ... 59

Lisa 4 – SUS küsimustik 66
Lisa 5 – Süsteemi vaadete kuvatõmmised 67

Jooniste loetelu

Joonis 1. Süsteemi arhitektuuri diagramm. 27
Joonis 2. Hoidlate vaade. 30
Joonis 3. Hoidla ülesannete vaade. 30
Joonis 4. Ülesande sisu vaade. 33

Tabelite loetelu

Tabel 1. Aurora süsteemi õigused. 31
Tabel 2. LLM raskusastme hindamise tulemused 51 eksamiülesandel..................... 39
Tabel 3. SUS küsimustiku tulemused... 49

1 Sissejuhatus

Aurora on Tallinna Tehnikaülikooli töötajatele mõeldud ülesannete haldamise register, kuhu koondatakse programmeerimisülesandeid erinevatest hoidlatest. Rakendus võimaldab kasutajatel ülesandeid vaadata ühest keskkonnast ning otsida neid erinevate parameetrite alusel. [1]

Aurora arendamise käigus on valminud neli lõputööd, mille raames on süsteemi täiendatud ja parendatud. Lisatud on turvafunktsionaalsusi ning parandatud otsingumootorit ja kasutajaliidest. Sellest hoolimata esineb rakenduses mitmeid puudusi, mis takistavad selle igapäevast kasutamist. Peamisteks probleemideks on vananenud versioonihaldus, aegunud raamistikud ning funktsionaalsuse puudumine, mis võimaldaks ülesannete kirjeldusi ja detaile vaadata otse Aurora rakenduses.

Lisaks esineb süsteemis mitmeid programmivigu. Näiteks ei ole GitLabi hoidlad ja Aurora rakendus omavahel sünkroonis: hoidla eemaldamisel GitLabist ei kajastu muudatus Aurora rakenduses. Samuti on GitLabiga esialgne sünkroonimine ajamahukas, kuna puudub võimalus hoidlaid ühekaupa valida ning automaatselt tõmmatakse alla kõik hoidlad korraga. Probleeme esineb ka rakenduse navigeerimises. [2]

Teema on aktuaalne, kuna igal aastal lisandub uusi üliõpilasi, kelle jaoks on vaja koostada uusi programmeerimisülesandeid, kontrolltöid ja eksameid. Õppejõudel on kaks valikut: kas luua täiesti uued ülesanded või kasutada olemasolevaid lahendusi erinevatest hoidlatest. Kui hoidlaid on väga palju, muutub sobivate ülesannete leidmine ajamahukaks ja ebaefektiivseks. Lisaks koostatakse eksami- ja kontrolltöövariandid sageli käsitsi, mis muudab protsessi aeganõudvaks ning raskendab ülesannete raskusastme ühtlast hindamist. See omakorda võib mõjutada hindamise objektiivsust, eriti suurte kursuste puhul. Varasemalt on loodud rakendusi, mis võimaldavad koostada teste kindla raskusastme ja ülesandetüüpide alusel tehisintellekti abil. Artiklis „Automated Test Creation Using Large Language Models: A Practical Application” jõuti järeldusele, et sellised süsteemid töötavad tõhusalt ning

vähendavad oluliselt õppejõudude töökoormust. [3]

Käesoleva töö eesmärgiks on luua uus terviklik ja kasutajasõbralik Aurora rakendus, mis vastab tänapäevastele arendus- ja turvanõuetele ning lahendab olemasoleva süsteemi peamised puudused. Selleks uuritakse olemasolevat Aurora süsteemi ja selle võimalusi, et selgitada välja, millised osad on vaja uues lahenduses säilitada ja millised ümber kujundada.

Töö käigus arendatakse uus rakendus, kus kirjutatakse ümber põhilised funktsioonid, kasutades tänapäevaseid tehnoloogiaid. Lisaks luuakse võimalus vaadata ülesannete failistruktuuri, parandatakse hoidlate võrdlemise süsteemi ning täiustatakse kasutajate õiguste haldust. Rakenduse arendamisel pööratakse erilist tähelepanu turvalisusele ning vastavusele Tallinna Tehnikaülikooli stiilinõuetele. Samuti disainitakse uus kasutajaliides, mis parandab süsteemi kasutusmugavust.

Lisaks tehnilistele parandustele käsitletakse töös ka eksamite ja kontrolltööde koostamise protsessi automatiseerimist. Selleks arendatakse süsteem, mis võimaldab hinnata ülesannete keerukust tehisintellekti abil ning genereerida automaatselt erineva raskusastmega töövariante.

Käesolev töö eristub varasematest lähenemistest, kuna olemasoleva rakenduse edasiarendamise asemel alustatakse Aurora arendamist uuesti. Töö käigus analüüsitakse, millised funktsionaalsused on süsteemi jaoks tegelikult vajalikud ning milliseid funktsionaalsusi uude lahendusse lisada ei ole otstarbekas.

Töö tulemusena valmib uus terviklik rakendus, mis sisaldab endas varasema Aurora süsteemi funktsionaalsusi uuendatud ja parandatud kujul. Uus rakendus järgib läbivalt turvanõudeid. Samuti on see kasutajasõbralik ning hõlmab endas uusi täiustavaid funktsionaalsuseid.

Lõputöö tellijaks on Tallinna Tehnikaülikooli Tarkvarateaduste instituut.

1.1 Metoodika

Töö esimeses etapis keskendutakse olemasoleva Aurora süsteemi analüüsile. Selle käigus hinnatakse süsteemi arhitektuuri, kasutuses olevaid tehnoloogiaid ning funktsionaalsusi, et selgitada välja, millised lahendused on mõistlik uude rakendusse üle viia ning millised tuleb ümber mõelda või uuest rakendusest välja jätta.

Saadud teadmiste põhjal arendatakse uus Aurora rakendus. Arendusprotsessis järgitakse agiilseid põhimõtteid, mis tähendavad pidevat iteratiivset arendust ja testimist ning võimaldavad süsteemi järk-järgult üles ehitada [4]. Funktsionaalsusi testitakse jooksvalt arenduse käigus, mis võimaldab tuvastada ja parandada vead kiiremini.

Töö käigus katsetatakse ka erinevaid tehisintellekti mudeleid, et hinnata nende sobivust programmeerimisülesannete keerukuse määramiseks ning eksami- ja kontrolltöövariantide automaatseks koostamiseks. Tulemusi võrreldakse ning valitakse lahendus, mis annab kõige usaldusväärsemaid tulemusi.

Arenduse käigus toimusid regulaarsed koosolekud tiimi ja juhendaja vahel, mille käigus arutati tehtud tööd, tekkinud probleeme ning edasisi arendussuundi. Igapäevaseks suhtluseks kasutati ka suhtluskanaleid, mis võimaldasid jooksvalt küsimusi lahendada ja ideid arutada.

1.1.1 Tehisintellekt

Töö tehnilise osa arendamisel kasutati abivahenditena ka erinevaid tehisintellekti mudeleid, näiteks Claude'i, ChatGPT-d ja Gemini't. Peamiselt kasutati Claude'i arendusetappide planeerimiseks, probleemide analüüsimiseks ning võimalike lahenduste läbimõtlemiseks enne nende realiseerimist.

ChatGPT-d ja Gemini't kasutati peamiselt vigade otsimiseks ja vajadusel ka nende lahendamiseks. Lisaks kasutati seda mõningate korduvate ja ajamahukate ülesannete automatiseerimiseks, näiteks failide ümberstruktureerimiseks või koodi ümbertõstmiseks.

2 Ülevaade probleemist ja eelnevast lahendusest

Käesolevas peatükis antakse ülevaade Aurora senisest ajaloost ja seisundist ning tuuakse välja süsteemi kitsaskohad, selgitatakse, miks pole vana süsteem jätkusuutlik ja vajab täielikku ümberkirjutamist ning millises mahus seda plaanitakse teha.

Esimeses jaotises analüüsitakse sarnaseid lahendusi. Teises jaotises kirjeldatakse varasemaid süsteemi arendusega seotud lõputöid, tuuakse välja olulisemad tähelepanekud ja töö tulemused. Kolmandas jaotises on järeldused varasematest töödest. Neljandas jaotises antakse ülevaade sellest, mida Aurora töö päriselt alguses endast kujutas. Lõpuks selgitakse, millised funktsioonid võetakse uude süsteemi üle ning millised eemaldatakse või asendatakse.

2.1 Sarnased lahendused

Autorid analüüsisid olemasolevaid lahendusi, mis võimaldavad programmeerimisülesannete haldamist, hoiustamist või automaatset hindamist. Analüüsi eesmärk oli välja selgitada nende süsteemide peamised funktsionaalsused ja kasutusvaldkonnad.

2.1.1 HackerRank

HackerRank on veebipõhine platvorm, mida kasutatakse programmeerimisoskuste arendamiseks, tehnilisteks tööintervjuudeks valmistumiseks ning kandidaatide hindamiseks [5]. Platvormi kasutavad nii tarkvaraarendajad oma oskuste harjutamiseks kui ka ettevõtted kandidaatide tehniliste teadmiste hindamiseks.

HackerRank sisaldab programmeerimisülesandeid erinevatest valdkondadest ning võimaldab lahendusi automaatselt hinnata. Lisaks pakub süsteem kandidaatide hindamise tööriistu, erinevaid petmise ja plagiaadi tuvastamise ning tööintervjuuks harjutamise võimalusi. Viimastel aastatel on platvormile lisatud ka tehisintellektil põhinevaid funktsionaalsusi.

Aurora ja HackerRank erinevad oma eesmärkide poolest märgatavalt. HackerRank kes-

kendub kandidaatide hindamisele ja tehniliste värbamisprotsesside abistamisele, Aurora eesmärk on koondada programmeerimisülesandeid erinevatest GitLabi hoidlatest ühte kesksesse registrisse. Aurora ei tegele kandidaatide hindamise ega tööintervjuude läbiviimisega.

Süsteemid lahendavad erinevaid probleeme, mistõttu ei saa Aurora asemel kasutada HackerRanki. Aurora ja HackerRank on sarnased eelkõige ülesannete raskusastmete ning ülesannete organiseerimise lahenduste poolest.

2.1.2 LeetCode

LeetCode on veebipõhine platvorm, mis on loodud programmeerimisoskuste arendamiseks ning tehnilisteks tööintervjuudeks ettevalmistamiseks [6]. Platvorm sisaldab suurt hulka programmeerimisülesandeid erinevatest valdkondadest, näiteks algoritmidest, andmestruktuuridest ja süsteemidisainist. Lisaks pakub LeetCode kasutajatele erinevaid mooduleid õppimiseks, mis aitavad keskenduda kindlate teadmiste ja oskuste arendamisele.

LeetCode'i üheks peamiseks tugevuseks on ülesannete kategoriseerimine ja filtreerimine. Ülesandeid on võimalik filtreerida erinevate teemade järgi, näiteks massiivide (*Array*), sõnede (*String*), graafide (*Graph*) või dünaamilise programmeerimise (*Dynamic Programming*) põhjal. Lisaks on ülesannetele määratud raskusastmed, mis võimaldavad kasutajatel valida oma teadmistele vastava keerukusega ülesandeid.

Auroraga võrreldes on LeetCode'i kõige sarnasemaks omaduseks ülesannete kategoriseerimine. Ka Auroras on võimalik ülesannetele lisada sildid, mis lihtsustavad ülesannete leidmist ja organiseerimist. Lisaks on mõlemal süsteemil olemas ülesannete raskustasemed. LeetCodes on raskusastmeteks lihtne, keskmine, raske. Auroras hinnatakse ülesannete raskusastet kümne palli skaalal.

Samas on süsteemide eesmärgid märkimisväärselt erinevad. LeetCode keskendub eelkõige programmeerimisoskuste arendamisele ning tööintervjuudeks ettevalmistamisele. Lisaks kasutavad ettevõtted platvormi kandidaatide tehniliste oskuste hindamiseks. Aurora eesmärk on koondada programmeerimisülesandeid erinevatest hoidlatest ühte kesksesse registrisse. Veel üheks suureks erinevuseks on see, et LeetCode'i ülesanded on loodud platvormi siseselt, kuid Auroras tulevad ülesanded GitLabi hoidlatest.

LeetCode erineb Aurorast märgatavalt, kuid mõlemas süsteemis on oluline roll ülesannete kategoriseerimisel ja leidmisel.

2.1.3 GitHub Classroom

GitHub Classroom on GitHubi poolt loodud keskkond õppetöö toetamiseks, mis võimaldab õppejõududel hallata programmeerimisülesandeid, kursuseid ja tudengite töid [7]. Süsteem on integreeritud GitHubi versioonihaldusplatvormiga ning võimaldab luua ülesandeid, jagada õppematerjale ja automatiseerida hindamisprotsessi.

GitHub Classroomi peamisteks funktsionaalsusteks on ülesandemallide loomine, automaatne hindamine, tagasiside andmine ning integratsioon erinevate õppehaldussüsteemidega. Ülesandemallide abil saavad õppejõud luua eeldefineeritud hoidlaid, mis sisaldavad ülesande kirjeldust, lähtekoodi, dokumentatsiooni ja muid õppematerjale. Ülesande avaldamisel luuakse igale õppijale või meeskonnale eraldi hoidla, mille kaudu toimub töö esitamine ja hindamine.

Aurora ja GitHub Classroom lahendavad osaliselt sarnast probleemi, kuna mõlemad võimaldavad programmeerimisülesannete hoiustamist ja haldamist. Samuti põhinevad mõlemad lahendused Git-põhistel hoidlatel.

Samas on süsteemide eesmärgid erinevad. GitHub Classroom keskendub peamiselt ülesannete jagamisele õppijatele ning nende lahenduste kogumisele ja hindamisele. Aurora eesmärk on pakkuda keskset programmeerimisülesannete registrit, kuhu koondatakse ülesandeid erinevatest hoidlatest. Süsteem võimaldab hallata hoidlaid gruppide kaupa, määrata kasutajatele grupisiseseid õiguseid ning otsida ülesandeid erinevate parameetrite alusel. Lisaks sisaldab Aurora funktsionaalsusi, mida GitHub Classroom ei paku, näiteks ülesannete automaatset raskusastme hindamist suurte keelemudelite abil.

GitHub Classroom pakub mitmeid kasulikke lahendusi ülesannete haldamiseks ja Git-põhiseks õppetööks, kuid selle peamine eesmärk on kursuste läbiviimine. Aurora keskendub seevastu olemasolevate ülesannete koondamisele, organiseerimisele ja taaskasutamisele, mistõttu ei saa Aurorat asendada GitHub Classroomiga.

2.2 Varasemad tööd

Aurora olemasolev versioon on välja arendatud nelja lõputöö raames. Järgnevalt on toodud kirjeldused igast eelnevast tööst, et selgitada Aurora arenduskäiku.

2.2.1 Süsteemi loomine

Esimene lõputöö „Programmeerimisülesannete haldamise register Aurora" valmis 2021. aastal. Töö fookuses oli tsentraalse hoidla loomine, et optimeerida õppejõudude ajakulu uute ülesannete ettevalmistamisel ja vanade süsteemsel säilitamisel. [1]

Enne töö alustamist analüüsiti, kuidas on sarnaseid eesmärke varasemalt lahendatud. Toodi välja, mis võimaldab ja mis takistab nende lahenduste kasutamist programmeerimisülesannete haldamiseks ülikoolis.

Töö eesmärk oli luua terviklik süsteem koos vajaliku töökeskkonna, liideste ja dokumentatsiooniga.

Töö autor jäi rahule loodud süsteemi üldise jõudluse, arhitektuurse laiendatavuse ning arendusprotsessi käigus saadud tagasisidele reageerimisega, kuid märkis, et mitmed komponendid jäid esialgsest skoobist välja. Olulisemate puudustena loetleti järgmised funktsionaalsused:

- programmeerimiskeelte tuvastamine faili sisust
- kasutaja autentimine ja autoriseerimine UniID abil
- administreerimistööriistad
- integratsioon Arete testimisteenusega
- proksi vastupidavuse parandamine

Kuna osa neist on süsteemi turvalisuse poolest hädavajalik, ei olnud võimalik selle töö tulemust kohe kasutusele võtta.

2.2.2 Õiguste ja statistika juurdearendus

Teine lõputöö „Programmeerimisülesannete haldamise registri Aurora õiguste ja statistika süsteemi juurdearendus" valmis 2022. aastal. Töö eesmärgiks oli lisada süsteemile puuduolevad funktsionaalsused ning täiustada olemasolevaid, et rakendus oleks päriselt

kasutatav. Esialgse lahenduse puudustena toodi välja kasutajate autentimise puudumine, programmeerimisülesannete statistika ülevaate puudumine ning kasutajaõiguste haldamise puudulikkus. [8]

Töö tulemusena realiseeriti funktsioneeriv kasutajate autentimise ja õiguste haldamise süsteem ning automatiseeritud statistika kogumine. Lisaks integreeriti Arete süsteem ning uuendati kasutatud tarkvaraversioone. Töö käigus avastati, et esirakenduse raamistiku versioonid olid aegunud, mis põhjustas palju probleeme ja lihtsalt uuendamine tundus liiga keeruline. Selle asemel tehti uus kasutajaliides, mille arendamisel võeti suund visuaalsele ühtsusele ülikooli teiste tarkvaralahendustega.

Positiivsena võib välja tuua, et kasutati agiilset tarkvaraarendusmeetodit, suheldi pidevalt juhendaja ja tellijaga. Samuti oli arendusprotsessis oluline meeskonnaliikmete vaheline pidev koostöö ning muudatuste põhjalik analüüs enne nende rakendamist. Testimine oli põhjalik ning hõlmas nii esi- kui ka tagarakendust, mis aitas tagada lahenduse kvaliteeti.

Töö autorid jäid rahule seatud eesmärkide täitmisega, märkides, et vaatamata kolmanda arendaja lahkumisele meeskonnast sujus koostöö hästi. Nende hinnangul muutus süsteem tänu tehnoloogilisele uuendusele kiiremaks ja kaasaegsemaks. Samas märgiti ka seda, et õiguste süsteemi oleks võinud teostada lihtsamalt. Toodi välja rakenduse aspektid, mis vajavad edasiarendamist:

- õiguste süsteemi edasiarendus ülesannete nähtavuse piiramiseks: õigused toimivad „kõik või mitte midagi” põhimõttel, mis osutub kriitiliseks puuduseks, kui rakendust kasutatakse mitme aine materjalide haldamiseks
- süsteemi muudatuste ja kasutajate tegevuste logimise lisamine
- ülesannete kommenteerimise funktsionaalsus
- kasutajaliidese edasiarendus: täpsemate veateadete lisamine, graafikute kuvamine ja muud väiksemad muudatused

Tellija kavatses siiski proovida järgneval semestril süsteemi oma õppeainetes rakendada.

2.2.3 Süsteemi stabiilsuse parendamine ja tehniline analüüs

Selgus, et Auroras on veel mitmeid vigu ning rakendus kasutusse ei jõudnudki. Kolmas lõputöö „Programmeerimisülesannete haldamise registri Aurora täiendused ja parandused" valmis 2023. aastal. Autorite hinnangul takistasid süsteemis esinevad kriitilised vead ja madal töökindlus selle praktilist rakendamist õppetöös. Eesmärgiks seati olemasoleva rakenduse analüüs, probleemide põhjuste tuvastamine ja parandamine. [9]

Peamiste probleemidena toodi välja, et süsteem on aeglane ja jookseb kokku. Lisaks esines puudusi statistika kogumisel ja kuvamisel, otsingufunktsionaalsuses, reageerimisvõimes ning hoidlate haldamisel. Töö eesmärgiks oli parandada süsteemi töökindlust ning tagada võimalus ülesannete otsimiseks, kommentaaride lisamiseks ja statistika kuvamiseks.

Lõputöö tulemusena paranes süsteemi töökindlus ning rakendust oli võimalik kasutada ilma tihedate katkestusteta. Täiendati statistika kuvamist, parandati otsingufunktsionaalsust, võimaldati kommentaaride lisamine ning parandati üldist jõudlust.

Töö käigus koostati põhjalik olemasoleva süsteemi analüüs ning toimus ka järjepidev koostöö tellijaga, mis võimaldas tuvastada ja prioritiseerida kriitilisemad probleemid. Samuti oli tugevuseks tehniliste arhitektuuripõhimõtete järgimine ja olemasoleva koodi refaktoreerimine.

Autorid tunnistasid töö kokkuvõttes, et oletus olemasoleva süsteemi parandamise lihtsusest võrreldes uue ehitamisega ei pidanud paika. Puudulik dokumentatsioon ja koodi keerukus tekitasid ettenägematuid takistusi, mis sundisid fookust pidevalt nihutama uute vigade parandamisele. Edasiste arendustena pakuti välja autentimine TARA (Riigi autentimisteenuse) kaudu, automaatne sildistamine ning väiksemad parandused süsteemi arhitektuuris.

2.2.4 Kasutajakogemuse parendamine

Neljas lõputöö „Programmeerimisülesannete haldamise registri Aurora kasutajakogemuse parendamine" valmis 2025. aastal. Autor valis selle teema, sest süsteemi senine kasutajaliides oli ebaefektiivne ja selle loogika takistas sujuvat tööd. Töö eesmärgiks on parandada Aurora kasutajakogemust, liidestamist Arete testimismootoriga ja süsteemi funktsionaalsuse ja kasutajamugavuse arendamist. [10]

Neljanda lõputöö tulemusena paranes Aurora eelkõige kasutusmugavuse ja töövoo ühtsuse osas. Süsteemi lisati ülesannete detailsem kuvamine, sealhulgas kirjeldused ja mallid, ning võimalus koodi otse Aurora keskkonnas testida ilma väliste tööriistadeta. Lisaks täiustati tulemuste kuvamist ning lisati selgemad teavitused kasutaja tegevuste ja süsteemi oleku kohta. Nende muudatuste tulemusel väheneks vajadus kasutada eraldi keskkondi ning kogu ülesande lahendamise protsess muutus ühtsemaks.

Autori hinnangul said püstitatud eesmärgid täidetud, kasutajaliides on muutunud mugavamaks, stabiilsemaks ja loogilisemaks. Edasiseks arenduseks pakkus ta välja statistika eksportimise, otsingu parendamise ja testide raskuse visualiseerimise.

2.3 Järeldused varasematest töödest

Analüüs näitab, et Aurorat on arendatud lühinägelikult, piirdudes vaid kõige kriitilisemate vigade parandamisega. Selline lähenemine on küll lisanud funktsionaalsust, kuid ei ole taganud süsteemi terviklikku ja jätkusuutlikku arengut, sest arhitektuursed ja kasutusloogika probleemid on endiselt jäänud püsima. Peamised probleemid on järgmised:

- **Kitsa skoobiga arendus ja integratsiooniprobleemid:** Iga töö keskendus konkreetse osa arendamisele, analüüsimata seda, kuidas uute lahenduste lisamine mõjutab süsteemi tervikuna. Seetõttu tõi iga uus funktsionaalsus kaasa ka uusi probleeme. Selle tagajärjel ei ole süsteemi eri osad omavahel ühtselt seotud.
- **Tehnilise võla kuhjumine:** Iga uus meeskond on pidanud ehitama eelmise autori loodud vundamendile. Kuna koodistiilid, arhitektuurilised otsused ja lahenduskäigud on olnud erinevad, on süsteemi kogunenud märkimisväärne hulk tehnilist võlga. See on viinud olukorrani, kus olemasoleva koodi muutmine on muutunud järjest keerukamaks ja ettearvamatumaks, mida kinnitasid ka 2023. aasta töö tulemused [9].
- **Dokumentatsiooni puudus:** Järjepidevuse puudumine arendajate vahel on viinud selleni, et süsteemil puudub terviklik dokumentatsioon ja ühtne kasutusloogika. See muudab uue arendaja jaoks süsteemi mõistmise aeganõudvaks, kuna suur osa energiast kulub koodist aru saamisele, mitte uue väärtuse loomisele.

Eelnevate tööde analüüs viitab sellele, et Aurora on arenenud erinevate autorite käe all ilma ühtse pikaajalise arhitektuurse plaanita. Järgmises jaotises antakse ülevaade süsteemi

tehnilisest seisukorrast autorite vaatlusel, et hinnata edasise arenduse perspektiive.

2.4 Olemasoleva süsteemi analüüs

Koodibaasi ja rakenduse testimise käigus analüüsiti süsteemi, et selgitada välja selle tugevused ja nõrkused. Analüüsi käigus selgus, et projekti versioonihaldus on ebaselge – põhiharu (*master*) ja viimase arendusetapi (*branch*) seisukord erinesid oluliselt, viidates puudulikule integratsiooniprotsessile. Kahe versiooni võrdlemisel ilmnesid järgmised asjaolud:

- **Kasutajaliidese regressioon:** Kuigi viimane töö keskendus kasutajakogemuse parendamisele, oli viimase etapi harus märgata visuaalseid regressioone. Näiteks olid varem loetavust parandanud kujunduselemendid (nagu tabeli päiste taustavärvid) kadunud. Samuti esines liideses ebaloogilisi kordusi – näiteks kuvati iga süsteemi lisatud hoidla kohta identne nupp *add to tagging queue*, mis ei olnud märgistatud ega seostatud konkreetse hoidlaga, muutes funktsiooni kasutamise segaseks.
- **Funktsionaalsuse lahknevus:** Põhiharus puudus võimalus failide sisu vaadata, pakkudes vaid linki välisesse GitLabi keskkonda. Viimase arenguetapi harus oli aga realiseeritud ülesannete kirjelduste kuvamine juhul, kui failinimed vastasid teatud reeglitele (`README.md` või `description.md`).

Hoolimata harude lahknevusest, tuvastati süsteemis teatud väärtuslikud funktsionaalsed osad, mida on otstarbekas uues lahenduses arvesse võtta.

2.4.1 Tuvastatud toimivad funktsionaalsused

Süsteemi analüüs tõi välja järgmised positiivsed aspektid ja edukalt realiseeritud osad:

- **Autentimine:** UniID kaudu sisselogimine töötab korrektselt ja stabiilselt.
- **Kommentaarid:** Võimalus lisada ülesannetele kommentaare on funktsionaalselt olemas.
- **Kirjeldused:** Ülesande kirjelduste kuvamine standardsetest failidest toimib valitud harus.
- **Sildistamine:** Olemasolev sildistamise süsteem on loogiline ning võimaldab ülesannete tõhusat filtreerimist.

- **Andmebaas:** Andmebaasi struktuur on üldiselt loogiline ja katab suurema osa süsteemi andmevajadusest.

2.4.2 Tuvastatud probleemid

Analüüs kinnitas, et süsteemsed vead ja tehniline võlg takistavad rakenduse laiemat kasutuselevõttu ja edasist hooldamist:

- **Vananenud tehnoloogiad:** Süsteemis kasutatakse vananenud tarkvaraversioone.
- **Platvormisõltuvus:** Failiteede käsitlemine ei ole platvormiülene (puudub `Path`-klassi kasutus ja normaliseerimine), mis tekitab probleeme erinevates serverikeskkondades.
- **Logimine ja veahaldus:** Logimine ja veateated ei ole piisavalt informatiivsed, et võimaldada tõhusat veatuvastust.
- **Koodi kvaliteet:** Koodibaas vajab põhjalikku korrastamist (palju kasutamata muutujaid, ebaselge ülesandega klassid).
- **Turvalisus:** Tuvastati kriitilisi turvapuudusi, sealhulgas API-pääsmete hoidmine otse lähtekoodis.
- **Sisu kuvamine:** Ülesandeid ei saa vaadata Auroras, vaid selleks peab minema GitLabi.
- **Kasutajakogemus:** Kasutajaliideses esineb ebaloogilisi elemente, ebaühtlast keelekasutust (eesti ja inglise keel segamini) ning süsteemi reageerimisvõime on puudulik.
- **Töövoo ebakõlad:** Õiguste süsteem ja navigatsioon ei ole optimaalsed ning automaatse sildistamise leht ei paku praegusel kujul selget kasu.
- **Ajalugu:** Sessiooni ajalugu ei ole informatiivne.

Nii varasemate analüüside kui ka käesoleva tehnilise auditi põhjal on selge, et Aurora süsteemi ei tasu enam parandada. Peamised tegurid, mis muudavad olemasoleva koodibaasi edasiarendamise ebaotstarbekaks, on järgmised:

- **Segadus arendusprotsessis ja versioonihalduses:** Viimase etapi arendused on jäänud põhiharusse liitmata ning on tekitanud visuaalseid ja funktsionaalseid regressioone. Puudub ühtne, usaldusväärne allikas (*source of truth*), mille põhjal arendust jätkata.
- **Arhitektuurne ebastabiilsus:** Projektiga on tegeletud mitme lõputöö raames ja iga kord jäid olulised probleemid tagaplaanile uute funktsionaalsuste kasuks, mis

ehitati niigi nõrgale vundamendile. Tulemuseks on süsteem, milles on palju probleeme ja vähe dokumentatsiooni. Ka 2023. aasta töö autorite kogemus näitas, et rakendus on jõudnud seisundisse, kus selle edasine haldamine ja silumine on täieliku ümberkirjutamisega võrreldes keeruline ja ebamõistlik [9].

- **Kriitilised turva- ja kvaliteedivead:** API pääsmete eemaldamine koodist ja ebakorrektsete lahenduste parandamine nõuaks nii fundamentaalseid muudatusi, et koodi refaktoreerimine tähendaks sisuliselt suure osa süsteemi ümberkirjutamist.

2.5 Uue lahenduse skoop

Aurorat on arendatud mitme erineva lõputöö raames, mille tulemusena on süsteemi funktsionaalsuste arv aastate jooksul märkimisväärselt kasvanud. Kõikide olemasolevate funktsionaalsuste üleviimine uude rakendusse oleks nõudnud oluliselt suuremat ajamahtu ning vähendanud võimalust keskenduda lahenduse kvaliteedile ja töökindlusele.

Seetõttu määrati koostöös töö tellijaga kindlaks uue süsteemi skoop. Eesmärgiks oli luua terviklik süsteem, mida oleks võimalik pärast arendust reaalselt kasutusele võtta ning millele saaks tulevikus täiendavaid funktsionaalsusi juurde arendada.

Kõige olulisemateks funktsionaalsusteks peeti autentimist, ülesannete kuvamist ja sarnasuse leidmist. Need moodustavad Aurora põhilise kasutusloogika ning on süsteemi igapäevase kasutamise seisukohalt kriitilise tähtsusega.

Lisaks soovis tellija lisada kaks uut funktsionaalsust: ülesannete raskusastme hindamine tehisintellekti abil ning automaatne eksamite ja kontrolltööde variantide koostamine. Nende funktsionaalsuste vajadus tuleneb sellest, et ülesannete arv kasvab igal aastal ning sobivate ülesannete käsitsi otsimine ja erinevate tööde koostamine muutub järjest ajakulukamaks.

Uuest lahendusest jäeti välja kommentaaride lisamise võimalus, detailne statistika, automaatne sildistamine. Need funktsionaalsused võivad olla kasulikud, kuid pole süsteemi esmase kasutuselevõtu seisukohalt kriitilised ning nende arendamine oleks suurendanud töö mahtu märkimisväärselt.

Valitud töömaht lasi keskenduda põhifunktsioonide kvaliteetsele loomisele ja panna aluse süsteemi edasisele arengule.

3 Loodava veebirakenduse ülevaade

Käesolevas peatükis antakse ülevaade loodava veebirakenduse nõuetest, kasutatud tehnoloogiatest ja peamistest funktsionaalsustest.

3.1 Nõuete määramine

Uue Aurora rakenduse funktsionaalsed ja mittefunktsionaalsed nõuded määratleti koostöös tellijaga, lähtudes varasemate lõputööde analüüsist, olemasoleva süsteemi puudustest ning uue lahenduse eesmärkidest. Nõuete määramise käigus keskenduti eelkõige nendele funktsionaalsustele, mis on vajalikud süsteemi reaalseks kasutuselevõtuks ning millele oleks võimalik tulevikus täiendavaid lahendusi juurde arendada.

Kõige olulisemateks funktsionaalseteks nõueteks peeti ülesannete kuvamist ja sarnasuse leidmist. Need moodustavad Aurora süsteemi põhiloogika. Lisaks sooviti uude süsteemi lisada tehisintellektil põhinevad lahendused, mis aitaksid vähendada aega, mida õppejõud kulutavad erinevate tööde koostamisele.

Funktsionaalsed nõuded:

- UniID-põhine kasutajate autentimine
- võimalus näha ülesannete failistruktuuri
- hoidlate võrdlemissüsteem
- hoidlate otsimise võimalus
- ülesannete võrdlemise süsteem
- tehisintellekt hindab ülesandeid
- automaatne eksamite või kontrolltööde koostamine

Lisaks funktsionaalsetele nõuetele täpsustati koostöös tellijaga ka süsteemi täiendavad tehnilised nõuded, et tagada Aurora terviklikkus.

Mittefunktsionaalsete ja täiendavate nõuetena määrati:

- olemasoleva kasutajate õiguste süsteemi ümberkujundamine ning rollipõhise õiguste halduse lisamine
- grupipõhise haldussüsteemi loomine, kus kasutajaid saab lisada gruppidesse ning gruppidele saab määrata rolle. Lisaks saab kasutajale soovi korral lisada veel eraldi õiguseid.
- võimalus lisada GitLabi pääsuvõti rakenduses
- kasutajaliidese optimeerimine ja reageerimisvõime lisamine

Arendustöös keskenduti eelkõige põhifunktsioonide kvaliteedile ja töökindlusele. Selline lähenemine võimaldas vältida varasemates töödes esinenud probleeme, kus süsteemi lisati küll palju funktsionaalsusi, kuid terviklik kasutajakogemus ja süsteemi töökindlus jäid nõrgaks.

3.2 Tehnoloogiate valik

Käesolevas jaotises antakse ülevaade kasutatud tehnoloogiatest. Loodava rakenduse tehnoloogiate valikul lähtuti olemasolevast Aurora arhitektuurist, varasemates töödes tehtud analüüsidest ning süsteemile seatud nõuetest. Eesmärgiks oli luua töökindel, pikaajaliselt hooldatav ja vajadusel laiendatav lahendus.

3.2.1 Tagarakendus

Tagarakenduse arendamisel otsustati kasutada samu tehnoloogiaid, mida oli kasutatud varasemates Aurora arendustes, ehk Java programmeerimiskeelt ja SpringBooti raamistikku. Varasemates töödes oli tehnoloogiate valik põhjalikult analüüsitud ja põhjendatud, mistõttu peeti mõistlikuks kasutada samu tehnoloogiaid. Kasutusele võeti uuemad tarkvaraversioonid, et tagada parem turvalisus, jõudlus ning pikaajaline jätkusuutlikkus.

Andmebaasina kasutati PostgreSQL-i, mis on relatsiooniline andmebaasihaldussüsteem. PostgreSQL valiti, kuna see oli kasutusel ka varasemates Aurora versioonides ning on laialdaselt kasutatav, töökindel ja hästi dokumenteeritud. Lisaks sobib relatsiooniline andmebaas hästi Aurora süsteemi jaoks, kuna süsteemis esineb palju omavahel seotud andmeid, näiteks kasutajad, ülesanded, grupid ja õigused.

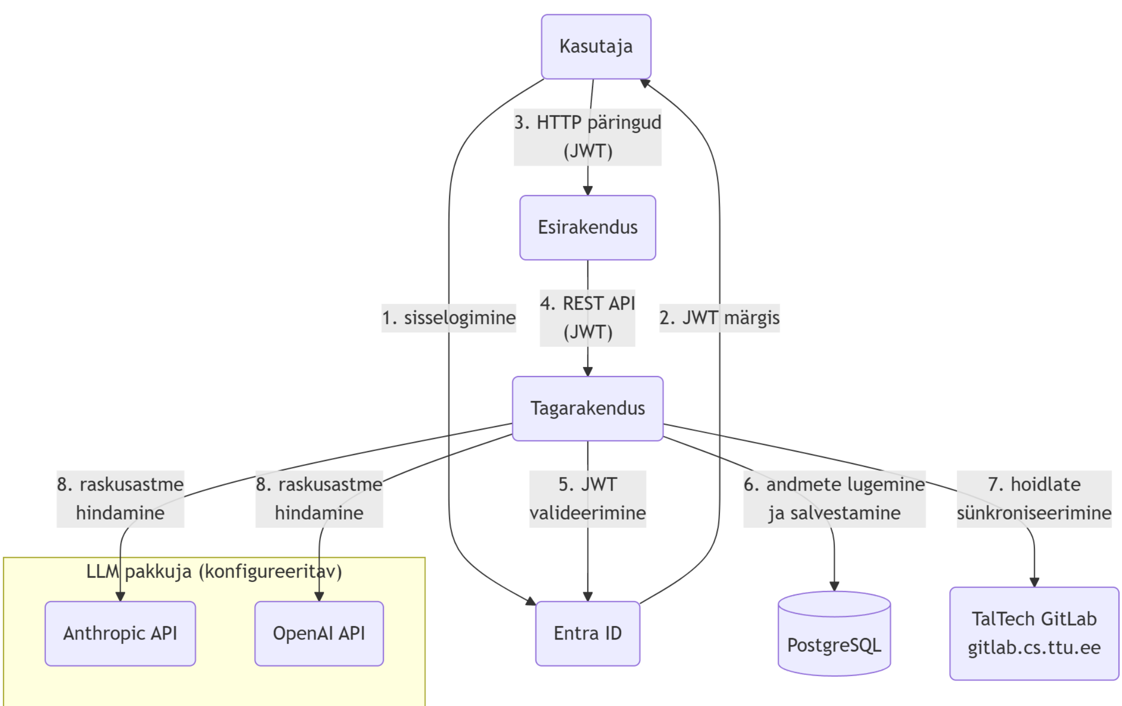
3.2.2 Esirakendus

Esirakenduse arendamisel valiti tehnoloogiateks React ja Vite. Reacti kasutamine tulenes eelkõige soovist Tallinna Tehnikaülikooli stiiliraamatut esirakenduse loomisel kasutada. Stiiliraamat pakub valmis kasutajaliidese komponente Reacti keskkonnas [11]. See võimaldas kiirendada arendusprotsessi ning tagada kasutajaliidese visuaalne ühtsus vastavalt ülikooli nõuetele. Vite valiti arendustööriistaks selle kiire arendus- ja ehitusprotsessi tõttu, mis parandab arendaja tööefektiivsust võrreldes traditsiooniliste lahendustega [12].

Programmeerimiskeelena kasutati TypeScripti, mis võimaldab tagada tüübikindluse ning vähendada vigade tekkimise tõenäosust arenduse käigus. See parandab koodi loetavust ja hooldatavust võrreldes JavaScriptiga [13]. Lisaks kasutati Node.js keskkonda esirakenduse arenduse toetamiseks ning Bootstrapi kasutajaliidese kujundamisel, et lihtsustada erinevate seadmetega kohanduvate ja standardiseeritud kasutajaliidese komponentide loomist [14].

3.3 Arhitektuur

Rakendus on üles ehitatud kahekihilise veebirakendusena. Esirakendus suhtleb tagarakendusega REST-rakendusliidese (REST API) kaudu kasutades autentimiseks Microsofti Entra ID pilvepõhist kasutajahaldusteenust koos kahe osapoole vahelise turvalise andmevahetusformaadiga (JWT ehk JSON Web Token). Tagarakendus on ühendatud Tallinna Tehnikaülikooli GitLabi serveriga. Kui eelmises Aurora versioonis oli rakendus rangelt just Tallinna Tehnikaülikooli GitLabi serveriga seotud, siis uues on võimalik lihtsalt kasutusele võtta ka teine GitLabi domeen. Selleks tuleb muuta vaid ühte keskkonna muutujat. Samuti on tagarakendus ühendatud PostgreSQL-i andmebaasiga, kuhu salvestatakse kõik rakenduse andmed. Täpsemat andmebaasi skeemi on näha Lisa 2 all. Ülesannete raskusastme automaatseks hindamiseks on tagarakendus integreeritud Anthropic Claude keelemudelite API-ga ning OpenAI API-ga. Rakenduse arhitektuur on üles ehitatud moodulitel, et kihid oleks selgelt eraldatud ja et süsteemi oleks lihtsam edasi arendada. Jooniselt 1 on näha täpsemat arhitektuuri ja kuidas süsteemi eri osad omavahel suhtlevad.



Joonis 1. Süsteemi arhitektuuri diagramm.

3.4 Veebirakenduse disain

Uue veebirakenduse disaini loomisel otsustati mitte kasutada varasemat esirakendust, kuna see ei vastanud enam süsteemi uutele nõuetele ega olnud kasutusmugavuse seisukohalt piisav.

Varasemas lahenduses esines mitmeid probleeme, näiteks ebaloogiline navigeerimine, ebaühtlane kasutajaliidese ülesehitus ning puudulik reageerimisvõime. Need tegurid raskendasid süsteemi kasutamist ja muutsid erinevate toimingute tegemise kasutaja jaoks keerulisemaks.

Uue esirakenduse loomisel seati eesmärgiks parandada kasutusmugavust, muuta süsteemi loogilisemaks ning tagada selle visuaalne ühtsus. Selleks kasutati Tallinna Tehnikaülikooli stiiliraamatut, mis võimaldas kasutada kasutajaliidese komponente, mis on kooskõlas ülikooli visuaalsete nõuetega.

Aurora esirakendus loodi Reacti raamistikku kasutades. Navigeerimine lahendati React Router teegi abil [15]. Tegemist on üheleherakendus (*Single Page Application*) lahendusega, mis tähendab, et vaadete vahel liikumisel ei ole vaja kogu lehte uuesti laadida [16]. Selline lähenemine lihtsustas rakenduse struktuuri ning võimaldas erinevate vaadete vahel liikuda

kiiremini ja sujuvamalt.

Navigeerimiseks loodi eraldi külgmenüü komponent, kuhu lisati rakenduse peamised vaated: töölaud, hoidlad, grupid, sildid ja Aurora tutvustuse leht. Menüü realiseeriti nii, et vaadete vahel liikumisel ei toimuks täislehe uuendamist. See parandab kasutuskogemust ning muudab rakenduse kasutamise kiiremaks ja mugavamaks. Lisaks aitas selline lahendus hoida rakenduse ülesehitust ühtlasena ning vähendada koodi kordumist erinevate vaadete loomisel.

Esirakenduse loomisel otsustati kasutada Tallinna Tehnikaülikooli stiiliraamatut, mis sisaldab valmis kasutajaliidese komponente, näiteks päiseid, nuppe, rippmenüüsid ja mitmeid teisi elemente. See aitas tagada kogu rakenduse ühtse visuaalse stiili ning vähendas vajadust luua eraldi disainikomponente. Stiiliraamatus olid määratletud ka rakenduses kasutatavad värvid ja kujunduspõhimõtted, mis lihtsustas esirakenduse kujundamist.

Kasutajaliidese loomisel pöörati tähelepanu sellele, et olulisemad tegevused ja nupud paikneksid kasutaja jaoks loogilistes kohtades ning süsteemi kasutamine oleks võimalikult lihtsasti mõistetav.

Rakenduse loomisel pöörati tähelepanu ka reageerimisvõimele. Väiksematel ekraanidel peidetakse külgmenüü automaatselt, et tagada parem kasutusmugavus mobiilseadmetes. Lisaks kasutati võimalikult palju Reacti ja Tallinna Tehnikaülikooli stiiliraamatu komponente, kuna need kohanduvad automaatselt erinevate ekraanisuurustega. Ka kõikide loodud komponentide puhul jälgiti, et need oleksid kasutatavad nii mobiiliseadmetes kui ka erinevate suurustega arvutites.

Lisaks on Aurorat võimalik kasutada nii inglise kui ka eesti keeles. Kõik kasutajaliidese tekstid määrati eraldi tõlkefailides, mis muudab hilisemate paranduste ja muudatuste tegemise lihtsamaks. Sellise lahenduse puhul ei ole vaja tekstimuudatuste tegemiseks otsida püsikodeeritud väärtusi erinevatest failidest, vaid sõnastust saab muuta tõlkefailides. Lisaks võimaldab selline lähenemine tulevikus rakendusele kergesti uusi keeli lisada.

3.5 Põhifunktsionaalsuste loetelu

Järgnevalt on esitatud Aurora olulisemate funktsionaalsuste detailne loetelu, kus iga alamkomponendi kohta on välja toodud selle lühikirjeldus. Kirjeldatud vaadete kuvatõmmised on esitatud lisas 5.

3.5.1 Autentimine Microsoftiga

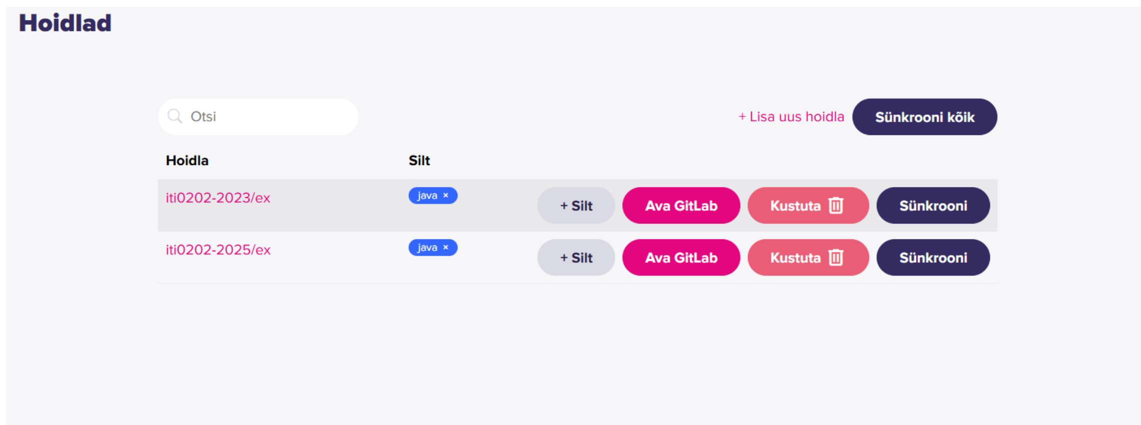
Rakendusse sisselogimine toimub Microsofti autentimisteenuse kaudu, kasutades Tallinna Tehnikaülikooli UniID kontot. Selline lahendus võimaldab tagab, et rakendusele pääsevad ligi ainult volitatud kasutajad.

Esmasel sisselogimisel salvestatakse kasutaja andmed automaatselt. Lisaks on kasutajal võimalik rakendusest välja logida ning vajadusel kontot vahetada.

3.5.2 GitLabi hoidlate tõmbamine

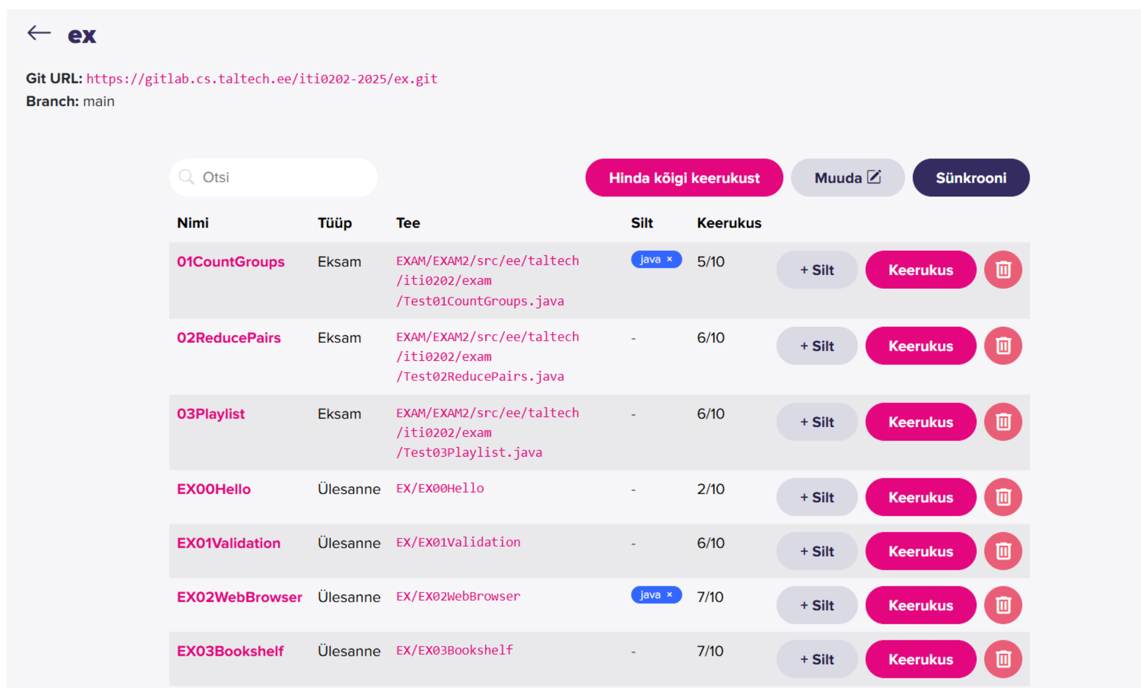
Üheks rakenduse põhifunktsionaalsuseks on GitLabist hoidlate tõmbamine. Selle toimimiseks peab kasutaja minema sätetesse, et lisada süsteemi enda GitLabi lugemisõigustega PAT (Personal Access Token ehk isiklik juurdepääsuvõti). See kõrvaldab vajaduse hallata Aurora jaoks eraldi GitLabi kasutajat ühe juurdepääsuvõtmega. Samuti piirab see kasutajale hoidlate tõmbamist, millele tal tegelikult ligipääsu ei tohiks olla.

Kui PAT on sätestatud, saab kasutaja lisada GitLabi hoidla Aurorasse kasutades valitud hoidla linki. Süsteem tõmbab hoidla alla. Juba alla tõmmatud hoidlaid on võimalik hiljem sünkroniseerida GitLabi staatusega, kasutades selleks eraldi nuppu hoidla nime juures. Hoidlate vaade on näha joonisel 2.



Joonis 2. Hoidlate vaade.

Joonisel 3 on kuvatud hoidla sisu vaade, kus on võimalik näha kõiki valitud hoidlas olevaid ülesandeid.



Joonis 3. Hoidla ülesannete vaade.

3.5.3 Õiguste ja gruppide süsteem

Rakenduses saab luua gruppe ning lisada nendesse kasutajaid erinevate rollide ja õigustega. Kõrgeima õigustega kasutaja on globaalne administraator (*Global Admin*), kes on ainus, kes saab luua uue grupi ja määrata sellele grupi administraatori (*Group Admin*). Mõlemal rollil on ka õigus gruppi kustutada.

Gruppi saab lisada liikmeid, kellele antakse vaikimisi assistendi roll (*Assistant*). Grupis on võimalik luua kohandatud rolle: igale rollile saab anda nime ja määrata sobivad õigused. Lisaks saab iga liikme õigusi individuaalselt kohandada: valides, milliseid õigusi ta võib või ei tohi omada, sõltumata tema rollist.

Gruppidega saab siduda hoidlaid. Üks hoidla saab kuuluda korraga vaid ühele grupile, kuid grupiga võib olla seotud mitu hoidlat. Grupiliikmed näevad automaatselt kõiki grupiga seotud hoidlaid ja nende sisu. Kasutaja näeb lisaks hoidlaid, mille ta on ise süsteemi lisanud, ning saab sünkroniseerida kõiki neile nähtavaid hoidlaid. Hoidla kustutamine süsteemist on lubatud ainult selle lisajale või globaalsele administraatorile.

Tabelis 1 on loetletud süsteemis olevad õigused koos kirjeldustega. Kasutaja õigused on alati seotud konkreetse grupiga, välja arvatud GLOBAL_ADMIN. Grupi administraatoril on vaikimisi kõik grupi taseme õigused; assistendil ei ole vaikimisi ühtegi.

Tabel 1. Aurora süsteemi õigused.

Õigus	**Kirjeldus**
`GLOBAL_ADMIN`	Täielik ligipääs süsteemile
`REPOSITORY_REGISTER`	Hoidlate lisamine ning nende sidumine grupiga
`GROUP_MANAGE`	Grupi ümbernimetamine
`MEMBER_MANAGE`	Grupis liikmete lisamine ja eemaldamine, rollide määramine liikmetele ja õiguste seadmine
`ROLE_MANAGE`	Grupis rollide loomine, muutmine ja kustutamine
`ASSIGNMENT_MANAGE`	Ülesannete märkimine, muutmine ja kustutamine grupiga seotud hoidlates
`EXAM_MANAGE`	Grupis töödevariantide koostamine, muutmine ja kustutamine

Antud süsteem on suur uuendus võrreldes eelmise Aurora versiooniga. Eelmises süsteemis olid grupid ainult inimestele rollide määramiseks ja puudus seos hoidlatega. Hoidla lugemisõigus ei määrata enam eraldi, vaid tuleneb gruppi kuulumisest.

Samuti muutus rollide loogika. Vanas süsteemis olid rollid globaalsed ehk roll andis õigused terve süsteemi ulatuses. Uues süsteemis kehtivad rollid vaid ühe grupi piires. Lisaks määratakse nüüd roll igale liikmele eraldi, mitte tervele grupile korraga. Veel saab kasutaja õigusi nüüd muuta nii, et selleks ei pea uut rolli looma.

Lisaks lihtsustati õiguste hulka. Eelmises süsteemis olid ette defineeritud 21 õigust, mis katsid kõigi objektidega seotud CRUD toimingud (andmebaasi põhioperatsioonid) eraldi. Uues süsteemis on vaid kuus fikseeritud õigust, mis keskenduvad sellele, mida kasutaja grupis teha saab.

3.5.4 Ülesannete märkimine

Pärast hoidla edukat sünkroniseerimist kuvatakse kasutajale hoidla struktuur interaktiivse failipuuna. Nii saab kasutaja valida kindlad kaustad või failid ning määrata need süsteemis programmeerimisülesanneteks. Süsteem toetab kahte tüüpi ülesannete liigitamist:

- **Tavaülesanne:** Mõeldud ülesannetele, kus iga ülesanne asub eraldi kataloogis ning kõik selles kataloogis leiduvad failid (kirjeldus, kood, testid) kuuluvad ühele ülesandele.
- **Eksamiülesanne:** See tüüp on loodud lahendama olukorda, kus mitu ülesannet on koondatud üheks eksamiks. Tüüpiliselt on sellistes kaustades iga ülesande testid eraldi failides, kuid kõigi ülesannete kirjeldused on koondatud ühte ühisesse juhendfaili (nt `README.md`).

Eksamiülesande märgistamisel genereerib süsteem iga valitud testifaili põhjal eraldiseisva ülesande. Seejuures liidetakse iga ülesandega automaatselt kaks ressurssi: konkreetne testifail ja ühine kirjeldusfail. Selleks, et kasutaja ei peaks lugema kogu eksami koondjuhendit ühe väikse ülesande juures, rakendab süsteem automaatset parsimist, mis püüab koondfailist eraldada just sellele testifailile vastava tekstiosa. Kui parsimine ei anna tulemust, teavitatakse kasutajat, et sisu vajab käsitsi kontrolli.

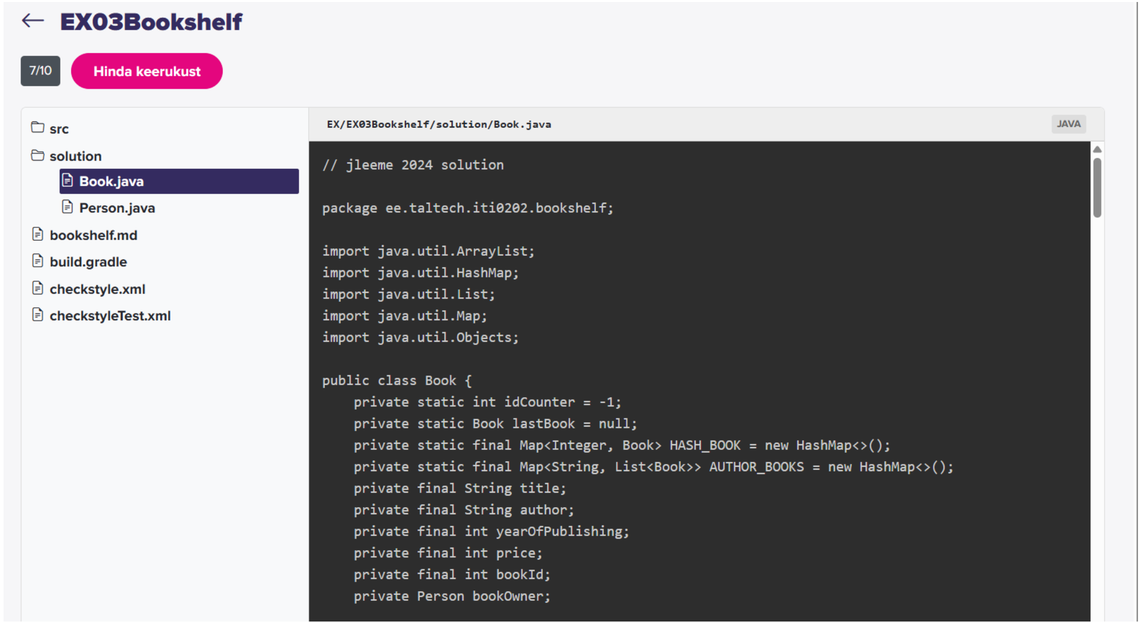
3.5.5 Ülesannete sisu kuvamine

Süsteem võimaldab ülesannetega seotud ressursse detailselt hallata ja vaadelda spetsiaalses ressursside vaates. Realiseeritud on intelligentne failitüüpide tuvastamine, mis eristab tekstifaile, lähtekoodi ja pildifaile, rakendades sisu kuvamisel vastavat kuvamisloogikat.

Iga ressursiga on seotud kategooria (*kirjeldus, test, mall, kood või muu*). Kuigi süsteem määrab esmase kategooria automaatselt faililaiendi ja nime põhjal, on kasutajal võimalus seda käsitsi muuta. Eksamiülesannete puhul pakub süsteem täiendavat paindlikkust: kui

automaatne parsimine ei suutnud kirjeldust korrektselt tuvastada, saab kasutaja süsteemi poolt pakutud teksti üle kirjutada või täiendada, tagades andmete korrektsuse registris.

Ülesande vaates kuvatakse ressursid failipuu kujul ning valitud faili sisu kuvatakse vaate parempoolses osas. Ülesande sisu vaade on joonisel 4.



Joonis 4. Ülesande sisu vaade.

3.5.6 Ülesannete võrdlus

Kuna programmeerimisülesandeid taaskasutatakse sageli aastast aastasse ja erinevatel kursustel, on süsteemi integreeritud võrdlusmootor, mis võimaldab selliseid ülesandeid tuvastada ning jälgida nende ajalist muutumist. Kattuvuste leidmiseks erinevates hoidlates vaatab süsteem ülesandeid tervikuna, võttes arvesse nii tekstilisi kirjeldusi kui ka kõiki kaasasolevaid faile.

Sarnasuse täpsemaks määramiseks jaotatakse need failid kategooriatesse, millest igaühte võrreldakse spetsiifilise, just selle failitüübi eripärasid arvestava meetodiga.

Analüüsi tulemusena kuvatakse kasutajale potentsiaalsed vasted. See lihtsustab materjalide haldamist olukorras, kus sama ülesande teisendid on hajutatud mitme eri hoidla vahel.

3.5.7 Sildistamissüsteem

Rakenduses on võimalik luua silte koos määratud nimetuse ja värviga. Silte on võimalik lisada hoidlatele ja ülesannetele, mida kuvatakse nende juures. See võimaldab kasutajal saada hea ülevaate hoidla või ülesande olemusest ilma, et oleks vaja nende sisusse rohkem süveneda.

Siltidle võib määrata ükskõik millise nimetuse vastavalt kasutaja vajadustele. Näiteks võib kasutaja nendega märkida ülesande programmeerimiskeele, teema või muu olulise tunnuse, mis aitab ülesandeid ja hoidlaid paremini kategoriseerida.

3.5.8 Keerukuse hindamine

Üheks tähtsamaks uuenduseks, mida eelmises Aurora süsteemis ei eksisteerinud, on ülesannete keerukuse hindamine suure keelemudeli (LLM) abil. See võimaldab kasutajal saada parema ülevaate hoidlates määratud ülesannete taseme kohta.

Kasutaja saab nupuvajutusega LLM-ile saata päringu, mis hakkab hindama valitud ülesande keerukust selle kirjelduse, seotud failide ja eri parameetrite põhjal. Parameetriteks on erinevad muutujad, näiteks kas ülesanne sisaldab pärilikkust või mitu testi ülesandel on. Võimalik on hinnata nii igat ülesannet eraldi kui ka korraga kõiki ühe hoidla ülesandeid.

Tavaliste ülesannete ja eksamite ülesannete keerukuste hindamised toimuvad erinevalt ning arvestavad antud ülesande tüübiga. Sama skooriga tavaülesanne ning eksamiülesanne ei ole reaalsuses sama keerukad: tavaülesanne vajab üldjuhul palju rohkem aega ning tuleb rohkemate tingimustega arvestada.

3.5.9 Töövariantide automaatne koostamine

Kasutajal on võimalik luua eksamiteks, kontrolltöödeks või tunnikontrolltöödeks mõeldud töövariante. Selleks on loodud automaatne süsteem, mis põhineb sildistamissüsteemil ning ülesannetele määratud keerukusel.

Ühe töö koostamiseks peab kasutaja määrama töö struktuuri ehk siltide järjendi. Esimene silt määrab esimese ülesande kategooria, teine teise ja nii edasi. Lisaks valib kasutaja, mitu töövarianti ta soovib ning millistest hoidlatest ülesandeid võtta. Tööde koostamine on

grupipõhine ehk valikus on vastava grupi hoidlad.

Peale valikute kinnitamist genereerib süsteem soovitud arvu töövariante täielikult erinevate ülesannetega. Edukaks kompileerimiseks peab valitud hoidlates olema piisavalt vastavate siltide ning määratud keerukustega ülesandeid. Töövariandid genereeritakse nii, et nende keskmine keerukus ei erineks rohkem kui ühe punkti võrra.

Hiljem on võimalik töövariante kustutada ning juurde lisada. Samuti saab üksikuid ülesandeid välja vahetada. Iga ülesande juures kuvatakse märge, mis näitab, kas ülesande silt või keerukus on vahepeal muutunud. See võimaldab kasutajal vajaduse korral ülesanne välja vahetada.

4 Veebirakenduse arendus

Käesolevas peatükis kirjeldatakse veebirakenduse arendusprotsessi ning loodud lahendusi keerukamatele funktsionaalsustele. Täpsemalt antakse ülevaade veebiteenuse-poolsest lahendusest, sealhulgas keerukuse hindamise loogikast, ülesannete sarnasuse kontrollimisest ning eksamiülesannete valimisest. Lisaks kirjeldatakse rakenduse testimist nii arenduse käigus kui ka hilisemates testimisetappides, et hinnata loodud lahenduse töökindlust ja korrektsust.

4.1 Veebiteenuse-poolne lahendus

Antud jaotises kirjeldatakse lahendusi keerukamatele funktsionaalsustele.

4.1.1 Autentimine Microsoftiga

Rakendus kasutab Microsoft Entra ID teenust kasutaja sisselogimiseks, kasutades ülikooli UniID-kontot [17]. Sisselogimise järel väljastatakse kasutajale allkirjastatud JWT, mille süsteem võtab vastu ja kontrollib selle autentsust. Vigase või puuduva märgi puhul ei lubata päringuid teha. JWT-märgist eraldatakse `oid` ehk kasutaja Entra ID objekti unikaalne identifikaator, ülikooli meiliaadress ning kasutaja nimi. Kui isik andmebaasis puudub, siis salvestatakse tema kohta uus kirje.

Peale isiku tuvastamist laetakse tema globaalsed õigused andmebaasist ja kogu info, sealhulgas kasutaja identifikaator ja nimi, salvestatakse `CustomAuthentication`-objekti. Seal olevad väärtused on eraldi abimeetodite kaudu ligipääsetavad kogu päringu vältel ning aitavad tuvastada kasutajat ning tema õigusi.

Sisselogimissüsteem põhineb vanas Aurora rakenduses kasutusel olnud lahendusel, kuid toob sisse mitmeid parandusi. Kasutaja andmebaasi identifikaator on nüüd kättesaadav autentimisobjektist, mis väldib korduvaid andmebaasipäringuid. Samuti on rangemalt piiritletud, millised domeenid tohivad tagarakendusega suhelda.

4.1.2 Ülesannete keerukuse hindamine

Ülesande keerukust otsustati hinnata LLM-iga. LLM-id on tehisintellekti põhised mudelid, mida on õpetatud läbi töötlema ja väljastama loomulikku teksti. Need mudelid on treenitud paljudel andmekogudel ning võimaldavad täita ülesandeid kiirelt ja tõhusalt, mis võtaks inimesel liiga palju aega [18].

Päringud on koostatud arvestades, et süsteemi kasutatakse põhiliselt algajatele mõeldud programmeerimiskursuste jaoks Pythoni ja Java keeles. Kuigi süsteem on suunatud nimetatud tüüpi kursustele, on siiski võimalik hinnata keerukust ka teist tüüpi ülesannete puhul, kuigi see ei pruugi nii täpne olla. Keerukuse hindamise protsess jaguneb sisuliselt neljaks osaks.

Esiteks loeb süsteem kõiki ülesandega seotud failiressursse ja jagab need järgnevatesse kategooriatesse:

- testifailid (sisaldavad sõna „*test*”)
- lahendusfailid (sisaldavad sõna „*solution*” või lühendit „*sol*”)
- mallifailid (sisaldavad kaks või enam „*TODO*” kommentaari)
- kõik ülejäänud koodifailid
- kirjeldust sisaldavad failid

Teiseks loob süsteem failide põhjal erinevad muutujad, kasutades regulaaravaldisi. On numbrilisi muutujaid, näiteks ridade arv kõikide failide lõikes ja testifailide arv. Samuti on Boole'i muutujaid, näiteks kas ülesanne kasutab pärilikkust või lambda-funktsioone.

Kolmandaks pannakse kokku prompt. Üks osa sellest on tekstifail, mis sisaldab:

- LLM-i rolli ülikooli tasemel programmeerimisülesannete eksperthindajana,
- detailset kirjeldust 10 palli keerukuseskaalast, millel on eraldi definitsioonid tavaülesannetele ning eksamiülesannetele,
- täpsustavaid meeldetuletusi ülesandetüübi, kursuse sisu ning muutujate puudumise kohta,
- näiteid päris ülesannetest ja nendele vastavatest skooridest.

Ülesannete näited ja määratud skoorid on koostatud treenimiseks programmeerimiskur-

sustega tihedalt seotud õppejõu poolt. Hiljem kasutatakse ka teisi õppejõu poolt esitatud ülesandeid ja skoore keelemudelite tulemuste hindamiseks. Veel antakse promptiga kaasa erinevad kogutud failid ja muutujad, sealhulgas kursuse kirjeldus, ülesande tüüp, ülesande kirjeldus, testid, lahendus ja teised. Sisendtokenite piiriks on vaikimisi määratud 6000 tokenit. Sellest tulenevalt võivad promptist välja jääda testid, ülesande mall ja ülesande lahendusfail. Ülejäänud andmed, sealhulgas ülesande kirjeldus, antakse alati promptiga kaasa. Kasutatud prompti saab täpsemalt uurida Lisa 3 all.

Neljandaks saadetakse prompt hetkel kasutuses olevale LLM-ile. Mudelile on antud käsk tagastada struktureeritud vastus:

- skoor (täisarv 1-10 vahel)
- mudeli enesekindlus hinnangu suhtes (*low*, *medium* või *high*)
- seletus (üks kuni kaks lauset põhjendades hinnangut ja välja tuues kõige mõjutavamad tegurid)
- üks kuni neli põhilist tegurit etteantud nimekirjast, näiteks *recursion*, *edge_cases* või *error_handling*

Tulemus salvestatakse andmebaasis vastava ülesande kirjele. Lisaks salvestatakse iga hinnang eraldi logimiseks mõeldud tabelisse, kus on ka märgitud hinnanguks kasutatud mudel, tokenite arv ja latentsusaeg.

Samuti seati eesmärgiks katsetada erinevaid LLM-e ning leida keerukuse hindamiseks parim mudel, arvestades täpsust, kiirust ja rahalist kulu 2026. aasta mai seisuga.

Täpsuse katsetamiseks hinnati 51 erinevat eksamiülesannet 2025. aasta Programmeerimise Algkursuselt, kus programmeeritakse Pythoni keeles. Tulemuste hindamiseks võeti n-ö õigeks skooriks õppejõu pandud hinnangud ja võrreldi neid keelemudelite saadud skooridega. Eelnimetatud eksamiülesandeid hinnati seitsme erineva LLM-ga. Esimese osa mudelitest moodustasid 2026. aasta mai seisuga Anthropicu firma Claude'i perekonna toetatud mudelid Claude Opus 4.7, Claude Sonnet 4.6 ja Claude Haiku 4.5 [19]. Teise osa mudelitest moodustas valik sama hetke seisuga OpenAI toetatud mudelitest: GPT-5.5, GPT-5, GPT-5 mini ja GPT-4.1 [20].

Tabelis 2 on välja toodud iga testitud mudeli kohta järgnevad mõõdikud:

- n ehk hinnatud ülesannete kogus
- MAE ehk keskmine absoluutne erinevus ennustatud ja tegeliku väärtuse vahel [21]
- %±1 ehk protsent hinnanguid, mis erinesid kuni ühe skooripunkti võrra tegelikust hinnangust
- ρ ehk Spearmani korrelatsiooni koefitsient, mis näitab, kui hästi kattuvad kahe andmestiku järjestused [22]
- D ehk dispersioon
- Kesk. kulu ehk keskmine kulu ühe päringu kohta USA dollarites
- p50 lat. ehk latentsusaja mediaan

Tabel 2. LLM raskusastme hindamise tulemused 51 eksamiülesandel.

Mudel	n	**MAE**	**%±1**	ρ	**D**	**Kesk. kulu**	**p50 lat.**
gpt-4.1	51	0,82	80,4%	0,784	1,69	$0,0080	6485ms
claude-opus-4.7	51	0,90	80,4%	0,827	3,70	$0,0390	4586ms
claude-sonnet-4.6	51	0,96	80,4%	0,789	2,74	$0,0172	4103ms
gpt-5-mini	51	1,20	68,6%	0,766	2,82	$0,0024	7997ms
gpt-5.5	51	1,22	64,7%	0,823	3,33	$0,0217	2974ms
gpt-5	51	1,24	60,8%	0,750	2,78	$0,0169	18044ms
claude-haiku-4.5	51	1,78	37,3%	0,785	1,94	$0,0058	2773ms

Mudelite hindamiseks kasutati 51 eksami ülesannet, mille raskusastmed jäid vahemikku 2–8 (keskmine skoor $\approx$ 5,8). Tabelist 2 on näha, et parimaks mudeliks osutus GPT-4.1, saavutades madalaima keskmise absoluutvea (MAE = 0,82) ning ennustades 80,4% juhtudest raskusastme, mis erines tegelikust kõige rohkem ühe punkti võrra (±1). Samal ajal oli see ka üks soodsamaid mudeleid (0,008 USA dollarit päringu kohta). Samasuguse ±1 täpsuse saavutasid ka Claude Opus 4.7 (MAE = 0,90) ja Claude Sonnet 4.6 (MAE = 0,96), kuid mõlemad on oluliselt kallimad. Paremusjärjestuse kvaliteedi poolest (Spearmani korrelatsiooni koefitsient) eristus Claude Opus 4.7 kõrgeima korrelatsiooniga (ρ = 0,827). See viitab heale võimele tuvastada ülesannete raskust teiste ülesannete suhtes. Ülejäänud neli mudelit jäid MAE poolest algtasemele. MAE $\approx$ 1,14 vastab igale ülesandele keskmise skoori panemisega. See jätab nende mudelite lisaväärtuse küsitavaks. Eriti silmapaistev oli Claude Haiku 4.5 nõrk tulemus (MAE = 1,78; ±1 täpsus vaid 37,3%).

4.1.3 Töövariantide automaatne koostamine

Tööde koostamise süsteem toimib viies etapis: sisendi valideerimine, sobivate ülesannete kogumine, juhuslik valik, keerukuse tasakaalustamine ning andmete salvestamine.

Teenus kontrollib esmalt, et kõik esitatud hoidlad kuuluvad päringus määratud gruppi ning et päritud sildid on andmebaasis olemas. Üks silt võib esineda mitu korda: näiteks struktuur `[„tsükkel", „tsükkel", „oop"]` on lubatud ja tähendab, et iga töövariant sisaldab kahte tsükliga seotud ülesannet.

Iga ülesandekoha jaoks päritakse andmebaasist kõik ülesanded, mis vastavad järgnevatele tingimustele:

- ülesanne asub mõnes valitud hoidlas
- ülesande tüüp on `EXAM`
- ülesandel on olemas keerukus
- ülesandega on seotud valitud silt

Kui vastavaid ülesandeid pole piisavalt, kuvatakse veateade täpse selgitusega.

Eksami reana salvestatakse juhuslik seeme (*seed*), mis võimaldab tulemuse vajadusel taastoota. Seejärel segatakse iga ülesandekoha jaoks kandidaatide nimekiri juhuslikus järjekorras ning valitakse sealt nii mitu ülesannet, kui on soovitud töövariante, tingimusel et ülesannet pole mõne teise ülesandekoha jaoks juba valitud. See tagab, et ükski ülesanne ei esine samas töös rohkem kui üks kord. Kui mõne ülesandekoha jaoks ei leidu piisavalt kasutamata kandidaate, siis katse tühistatakse ja alustatakse uuesti. Pärast 200 ebaõnnestunud katset tagastab süsteem vea.

Kokkupandud töövariantide jaoks arvutatakse iga versiooni keskmine keerukus ning kontrollitakse, kas kõrgeima ja madalaima keskmise vahe jääb ühe punkti piiresse. Kui tingimus pole täidetud, käivitatakse kuni 50 kordusega vahetustsükkel: leitakse kõrgeima ja madalaima keskmise keerukusega versioon ning ülesandekoht, kus nende kahe versiooni ülesannete vahetus annaks suurima paranemise. Kui keerukusvahe ei jõua 50 vahetusega ühe punkti piiresse, katse tühistatakse ja proovitakse uuesti. Kokku tehakse kuni 200 katset, enne kui süsteem annab vea.

Õnnestumise korral salvestatakse koostatud töö ja kõik tema versioonid. Iga tööülesande jaoks salvestatakse ka koopia tema sildist ja keerukusest. Seda kasutatakse töö lugemisel kõrvalekallete tuvastamiseks. Kui silt või keerukus on vahepeal muutunud, tagastatakse selle kohta info, et kasutaja saaks vajadusel ülesande välja vahetada.

4.1.4 Sarnasus

Vanas süsteemis oli sarnasuskontrolli peamiseks takistuseks ebaefektiivne algoritmivalik ja filtreerimise puudumine. Kõikide failide tähemärgipõhine võrdlemine Levenšteini algoritmi abil viis tundidepikkuse tööajani ja genereeris palju müra, leides sarnasusi ühe hoidla siseselt. Uues süsteemis optimeeriti protsessi kolmel tasandil: ülesannete filtreerimine, sisu kategoriseerimine ja spetsiifiliste algoritmide rakendamine vastavalt faili tüübile.

Esmalt välistati hoidlasisene võrdlus, keskendudes vaid erinevate hoidlate vahelisele sarnasusele. Kiiruse tõstmiseks katsetati Jaccardi sarnasuse hindamist MinHash-algoritmiga.

MinHash on algoritm, mis võimaldab umbkaudselt määrata Jaccardi sarnasust, ilma et peaks kulutama aega kogu hulga võrdlemisele. Sarnasuse leidmiseks tehakse mõlemast tekstist k-sümboli pikkuste sõnede hulk (*k-shingling*). Hulkade otse võrdlemise asemel räsitakse elemendid ära ja genereeritakse nendest signatuur. Selline hinnang põhineb seosel, et tõenäosus, et kahe hulga signatuurid kattuvad, on võrdne nende hulkade Jaccardi sarnasusega. [23]

Jaccardi sarnasus on meetod kahe hulga sarnasuse mõõtmiseks. See arvutatakse ühiste sõnade hulga ja kõigi unikaalsete sõnade hulga jagatisena [23]. Kuigi Jaccardi sarnasus on teoreetiliselt hulkade põhine, tekitab *k-shinglingi* kasutamine sõltuvuse sümbolite järjekorrast, mis muutis meetodi tundlikuks struktuursetele muudatustele. Kui testides muudeti meetodite järjekorda või ülesande kirjelduses tõsteti lõike ümber, langes sarnasuse skoor märgatavalt, kuigi sisu jäi samaks. Seetõttu jäi MinHash esmaseks variandiks ning MinHashile järgneb täpsem analüüs.

Katsetamise käigus jõuti järeldusele, et ei saa võrrelda kõiki faile sama algoritmiga, kuna muudatuste olemus on erinev (kirjelduses muudetakse sõnastust, testides struktuuri). Ressursi olemile lisati tüübi väli, millega määrati, kas tegemist on kirjelduse, testi, koodifailiga või muu failiga, mis pole sarnasuste leidmiseks oluline. Edasi oli võimalik eri tüüpi failidele

rakendada eri algoritmi.

Ülesannete tekstiliste kirjelduste võrdlemiseks analüüsiti kolme levinud algoritmi, et leida sobivaim lahendus ülesannete variatsioonide ja korduvkasutuse tuvastamiseks.

- **Jaccardi sarnasus (MinHashiga):** Eelnevalt läbi katsetatud variant.
  - väga kiire
  - tundlik teksti ümberstruktureerimise suhtes
- **Koosinussarnasus:** Teksti vaadeldakse vektorina n-mõõtmelises ruumis, kus koordinaadid tähistavad sõnade esinemissagedust. Sarnasus defineeritakse kui kahe vektori vahelise nurga koosinus. [24]
  - Arvestab sõnade esinemissagedust.
  - Teksti ümbertõstmine ei mõjuta tulemust.
  - Lühemates tekstides muutuvad esinemissagedused drastilisemalt ja see mõjutab tulemust.
- **Muutmiskaugus *(Edit distance)*:** Dünaamilisel planeerimisel põhinev algoritm, mille eesmärk on leida minimaalne arv lubatud operatsioone, et muuta üks sõne teiseks. [25]

  Käesolevas töös kasutati sarnaselt vanale süsteemile Levenšteini kaugust, kus lubatud on kolm operatsiooni: sümboli lisamine, kustutamine ja asendamine. [26]
  - Leiab sarnasused sümboli täpsusega, mis sobib hästi, kui tekst on lühike.
  - pikkade tekstide jaoks liiga ajamahukas
  - väga tundlik teksti ümbertõstmise suhtes

Analüüsi tulemusena otsustati uues süsteemis rakendada kombineeritud lahendust, mis kasutab ära erinevate algoritmide tugevusi:

- Lühikeste tekstide (kuni 150 sümbolit) sarnasust hinnatakse Levenšteini algoritmiga.
- Pikemaid tekste võrreldakse koosinussarnasuse algoritmiga.

Koodi sarnasuse tuvastamine osutus märksa keerukamaks kui tavaline tekstide võrdlemine. Esmalt uuriti juba eksisteerivaid tarkvaralisi lahendusi, millest lähemalt analüüsiti tunnustatud plagiaadituvastusvahendit JPlag. Nimetatud tööriist genereerib koodist märgiste jadasid (*tokens*), ignoreerides identifikaatoreid (muutujate ja meetodite nimesid) ning keskendudes programmi struktuursele loogikale. JPlagi põhiidee seisneb esituste paarikaupa võrdlemises,

mis võimaldab kogu valimi põhjal tuvastada statistilisi anomaaliaid [27].

Kuigi see on efektiivne üliõpilaste lahenduste võrdlemisel, osutus see ebaefektiivseks ülesannete testfailide analüüsimisel. Kuna testid on kirjutatud struktuurselt väga sarnaselt (kasutatakse samu annotatsioone, sarnaselt defineeritud meetodeid ja kohustuslikke *assert*-lauseid lõpus), leidis JPlag kohati, et ülesande testfail on sarnasem mõne täiesti teise ülesande testiga kui sama ülesande teise versiooniga. Lisaks ei sobi klassikalised plagiaadituvastuse algoritmid Aurorasse seetõttu, et süsteemis võrreldakse korraga vaid kahte konkreetset ülesannet, mis tähendab, et puudub laiem taustsüsteem ülejäänud paaride sarnasuse kõrvalekallete tuvastamiseks.

Kuna sarnaste ülesannete tuvastamisel on olulised identifikaatorid ka klasside, funktsioonide ja meetodite nimed, otsustati parsida koodifailidest olulised elemendid ja nende esinemissagedused ning teostada võrdlus nende põhjal.

Eri tüüpi koodifailidest eraldati spetsiifilised tunnused, mida võrreldi failitüübile sobivate algoritmidega:

- **Mallid:**
  - Koodist eraldati funktsioonide, klasside ja meetodite nimed.
  - Kuna need elemendid on unikaalsed, arvutati hulkade sarnasust Jaccardi indeksi abil.
- **Testid:**
  - Lähtekoodist tuvastati testimeetodite nimed ja nende poolt välja kutsutavad meetodid.
  - Andmetest filtreeriti välja testiraamistikele tüüpilised standardmeetodid (näiteks `assertEquals`, `assertTrue`).
  - Sarnasuse leidmiseks kombineeriti Jaccardi indeksit kattuvuse kordajaga (*overlap coefficient*). Selle mõõdiku matemaatiline loogika ja kasutus kooditunnuste võrdlemisel tugineb andmeteaduse kirjandusele [28].
  - Kattuvuse kordaja võimaldab hinnata, kas üks testide hulk on teise alamhulk, tuvastades efektiivselt ülesanded, mis on suurel määral täiendatud. Näiteks kui vanas versioonis oli 10 testi ja uues tehti 10 juurde, siis Jaccardi indeksi väärtuseks kujuneb $0{,}5$, kuigi tegelikult sisaldub terve vana versioon uue sees.

Kattuvuse kordaja väärtus oleks sellisel juhul aga $1{,}0$.

  - Siiski ei ole ka kattuvuse kordaja omaette efektiivne. Kui väiksem hulk sisaldab tüüpilisemaid elemente, võib suur osa sellest sisalduda suuremas hulgas, ilma et tegemist oleks päriselt sarnaste hulkadega. Kuna selline lähenemine ei võtaks arvesse alamhulka ümbritsevat konteksti, otsustati tulemuste tasakaalustamiseks mõlemat mõõdikut kombineerida.
  - Selline hübriidne lähenemine võimaldas tuvastada struktuurset sarnasust ka olukordades, kus ülesande edasiarenduse või muutmise käigus oli teste märgatavalt lisatud või eemaldatud. Kahe mõõdiku koos rakendamine ja seeläbi üheaegne globaalse sarnasuse (*resemblance*) ning osalise sisalduvuse (*containment*) hindamine tugineb Broderi matemaatilisele teooriale [29].
- **Muu kood:**
  - Koodist eraldati funktsioonide, klasside ja meetodite nimed koos nende esinemissagedusega.
  - Filtreeriti välja programmeerimiskeeltele omased rutiinsed baasfunktsioonid (näiteks `equals`, `print`, `len`).
  - Kuna andmed sisaldasid elementide esinemiskordi (sagedusvektoreid), arvutati sarnasust koosinussarnasuse meetodiga.

4.1.5 Eksami ülesannete valimine

Vanas süsteemis toimus ülesannete genereerimine automaatselt hoidla sünkroniseerimise järel. See tõi kaasa olukorra, kus süsteem märkis ülesanneteks ka mitterelevantseid faile (nt pildid) ilma võimaluseta vigu parandada. Uues süsteemis asendati see loogika käsitsi ülesannete määramisega, mis annab kasutajale kontrolli andmete struktuuri üle. Kasutaja saab valida, kas ülesanne on mõeldud ühe tervikuna (tavaülesanne) või mitme ülesande kollektsioonina (tunnikontrollid, kontrolltööd, eksamid). Eristamine on vajalik, kuna eksamite puhul taaskasutatakse sageli üksikuid ülesandeid ja sarnasuse analüüsis on oluline võrrelda ülesandeid eraldiseisvalt, mitte terve eksami kaupa.

Kuna ülesannete failide nimetamisel ja kataloogistruktuuris puudus ühtne standard, osutus kõige efektiivsemaks meetodiks vaadelda iga testifaili kui ülesannet.

- **Tuvastamine:** Süsteem tuvastab testifailid nime järgi (kokkuleppeliselt algab failinimi

sõnaga „*test*”).

- **Seostamine:** Igale testile lisatakse juurde üldine kirjeldusfail (eeldatavalt `README.md`, `description.md` või `exam.md`), mis sisaldab kogu eksami juhendit. Ülejäänud faile ignoreeritakse.

Eksamiülesande kirjelduse kättesaamiseks kirjutati algoritm, mis püüab pealkirjade struktuuri abil leida konkreetse ülesandega seotud lõiku. Protsessi käigus selgus aga, et rangete reeglite puudumine tekstifailide vormistamisel muutis automaatse parsimise ebakindlaks järgmiste probleemide tõttu:

- **Struktureerimata mallid:** Kuna mall sisaldab informatsiooni kõigi ülesannete kohta, on vaja see algoritmiga eemaldada. Seda teeb aga keeruliseks ühtse struktuuri puudumine. Mallile võib eelneda pealkiri, aga ei pruugi. Samuti esines kohati enne malli selle kohta käiv lause.
- **Hierarhiline ebakõla:** Eri eksamivariantides kasutati pealkirjade tasemeid (*header levels*) ebaühtlaselt. Näiteks võisid ülesandega seotud klasside nimed olla vormistatud sama taseme pealkirjana kui ülesande enda nimi.
- **Inimfaktor ja ebatäpsused:** Nummerdusvead, näpuvead ülesannete nimedes ning lahknevused testifaili nime ja kirjelduses toodud pealkirja vahel muutsid regulaaravaldistel põhineva tuvastamise ebatäpseks.

Analüüsi tulemusena jõuti järeldusele, et kõigi võimalike piirjuhtude algoritmiline lahendamine ei ole mõistlik. Selle asemel rakendati poolautomaatne lähenemine: süsteem teeb esmase parsimise parima oletuse põhjal, kuid juhul kui selle tulemus pole rahuldav, saab kasutaja käsitsi õige kirjelduse salvestada.

4.2 Testimine ja valideerimine

Selles jaotises kirjeldatakse rakenduse testimise protsessi ning kasutatud meetodeid. Testimise eesmärk oli veenduda, et autorite loodud lahendus töötab korrektselt ja vastab seatud nõuetele. Testimine toimus järk-järgult kogu arendusprotsessi vältel.

4.2.1 Testimine tellija esindajaga

Lõputöö tellijaks oli Tallinna Tehnikaülikooli Tarkvarateaduste instituut. Tellija esindajaga toimusid iganädalased koosolekud, kus autorid näitasid arenduse hetkeseisu, tutvustasid oma lahendusi ja ideid ning valminud funktsionaalsusi. Koosolekutel saadud tagasiside põhjal said autorid funktsionaalsusi täiendada vastavalt tellija ootustele ja nõuetele.

Selline lähenemine võimaldas võimalikke probleeme ja puudujääke tuvastada juba arenduse käigus. Lisaks aitas pidev tagasiside tagada, et arendatav süsteem vastaks võimalikult hästi lõppkasutaja vajadustele.

4.2.2 Testimine arenduse käigus

Tagarakenduse testimiseks kasutati mitmeid erinevaid meetodeid. Funktsionaalsuste korrektsuse kontrollimiseks koostati 596 üksustesti, mis katavad kõik rakenduse funktsionaalsused. Üksustestide raames kontrolliti ka erinevate päringute edukat toimimist ning süsteemi poolt tagastatavate HTTP staatusekoodide vastavust ootustele.

Lisaks üksustestidele testiti funktsionaalsuste toimimist ilma turvafiltriteta, et hinnata äriloogika toimimist sõltumatult turvakihist. See võimaldas äriloogikat ja turvalisust eraldi ning põhjalikumalt kontrollida.

Samuti testiti kõiki funktsionaalsusi manuaalselt. Manuaalselt kontrolliti ka kasutajaõigustega seotud piiranguid: veenduti, et globaalse administraatori õigustega kasutaja pääseb ligi kõikidele süsteemi funktsionaalsustele, samas kui tavakasutaja saab kasutada üksnes neid funktsionaalsusi, milleks talle on õigused antud.

Esirakenduse testimine toimus pidevalt kogu arendusprotsessi vältel. Automaattestide kasutamisest loobuti teadlikult, kuna kasutajaliides ja selle funktsionaalsused muutusid arenduse käigus sageli. Nende pidav uuendamine oleks olnud ajakulukas ja ebaefektiivne. Seetõttu eelistati manuaalset testimist.

Iga funktsionaalsus testiti kohe pärast selle realiseerimist. Kontrolliti nii tavapäraseid kasutusstsenaariumeid kui ka äärmuslikke olukordi, nagu vigaste sisendandmete esitamine ja kohustuslike väljade tühjaks jätmine.

Suuremate muudatuste ja uute funktsionaalsuste lisamisel kontrolliti üle ka olemasolevad lahendused, et teha kindlaks, kas varasemad funktsionaalsused töötavad endiselt korrektselt. Enne koodi mestimist põhiharusse viisid testimist läbi ka teised tiimiliikmed, mis võimaldas avastada probleeme, mis arendajale endale ei pruukinud koheselt märgatavad olla, ning andsid tagasisidet nii kasutusmugavuse kui ka töökindluse kohta. Tiimikaaslaste leitud vead sai kiiresti parandada, mis aitas vältida nende kuhjumist arenduse hilisemas faasis.

4.2.3 Testimine Thymeleafi abil

Tagarakenduse arendamise etapis kasutati funktsionaalsuste valideerimiseks ja visuaalseks testimiseks Thymeleaf-mallimootorit [30]. Kuna tegemist on serveripoolse tehnoloogiaga, võimaldas see vahetult kontrollida andmete liikumist ja algoritmide väljundit, ilma et oleks pidanud ootama esirakenduse täielikku valmimist.

Selline lähenemine pakkus arendusprotsessis mitmeid eeliseid:

- **Kontseptsiooni tõendus (*Proof of Concept*):** Thymeleafi abil loodud vaated võimaldasid varajases etapis veenduda, et planeeritud süsteemiarhitektuur on teostatav ning tuvastada tagarakenduses vajalikud komponendid reaalsete andmete kuvamiseks.
- **Silumine:** Koodi oli võimalik valideerida mitte ainult testide kirjutamisega, vaid ka kasutaja rollis.

Valminud prototüüp tõestas süsteemi toimivust ja andis suuna kasutajaliidese kujundamiseks.

4.2.4 Tiimisisene testimine

Rakenduse testimine toimus töö autorite poolt jooksvalt kogu arendusprotsessi vältel. Arenduse käigus testiti esi- ja tagarakendust peamiselt eraldi. Esirakenduse puhul kontrolliti, kas lisatud funktsionaalsused töötavad ootuspäraselt ning kasutajaliides kuvab andmeid korrektselt. Tagarakenduse testimisel kasutati Thymeleafi, mille abil kontrolliti rakenduse loogika ja andmete töötlemise korrektsust.

Arenduse lõppfaasis testiti rakendust ka tervikliku süsteemina. Kõik tiimiliikmed kontrollisid funktsionaalsusi, et veenduda nende korrektses toimimises ning kontrollida, kas esirakendus saab tagarakenduselt õiged andmed kätte. Lisaks hinnati kasutajamugavust,

võimalikke veateateid ning otsiti olukordi, kus rakendus ei pruugi käituda ootuspäraselt.

Tiimisisene testimine võimaldas avastada probleeme erinevatest kasutajaperspektiividest. Testimise käigus koguti kokku tiimiliikmete kommentaarid ja tähelepanekud, mille põhjal analüüsiti, kas tegemist oli tegelike vigadega või rakenduse ootuspärase käitumisega ning kuidas saaks rakendust täiustada. Saadud tagasiside põhjal viisid töö autorid enne töö valmimist sisse vajalikud parandused ja viimased täiendused.

4.2.5 Valideerimine

Töö lõppfaasis viidi läbi süsteemi kasutatavuse testimine ühe õppejõu ning nelja abiõppejõu seas. Testimise alguses anti osalejatele globaalse administraatori õigused ning tutvustati lühidalt süsteemi eesmärki ja peamisi funktsionaalsusi. Seejärel paluti testijatel rakendust iseseisvalt kasutada ning tutvuda selle erinevate võimalustega.

Testimise käigus paluti osalejatel täita SUS küsimustik, mis koosnes kümnest küsimusest. Küsimustiku näidis on olemas Lisas 4. Iga küsimuse puhul tuli vastata viiepallisel skaalal. Küsimustik koostati *System Usability Scale* (edaspidi SUS) metoodika põhjal.

SUS on laialdaselt kasutatav küsimustiku tüüp süsteemide kasutatavuse hindamiseks. Meetodi töötas välja John Brooke 1996. aastal. SUS-i eesmärk on pakkuda kiiret ja lihtsat viisi süsteemi kasutuskõlblikkuse mõõtmiseks. SUS küsimustik koosneb kümnest väitest, millele vastatakse viiepallisel skaalal. Brooke on välja toonud, et vastaja peaks andma vastused võimalikult kiiresti, ilma üksikute küsimuste üle liigselt mõtlemata, sest tavaliselt näitab esmane reaktsioon kõige paremini kasutaja tegelikku arvamust. [31]

Bangor, Kortum ja Miller uurisid SUS-i tulemuste tõlgendamist ning nende seost kasutajate üldiste hinnangutega süsteemi kasutatavusele. Autorid analüüsisid varasemate uuringute SUS tulemusi ning võrdlesid neid omadussõnaliste hinnangutega (*adjective ratings*). Uuringu tulemusena leiti, et SUS skooride ja kasutajate subjektiivsete hinnangute vahel esineb tugev seos, mis võimaldab tulemusi tõlgendada erinevate kasutatavuse kvaliteedikategooriate kaudu. Autorite hinnangul muudab selline lähenemine SUS tulemused lihtsamini mõistetavaks ning aitab paremini selgitada, mida konkreetne skoor süsteemi kasutatavuse kohta näitab. [32]

SUS küsimustiku tulemuseks saadakse üks koondskoor, mis kirjeldab hinnatava süsteemi üldist kasutatavust. Küsimuste tulemusi ei tõlgendata eraldi, vaid need kasutatakse lõpliku skoori arvutamiseks. Skoori leidmiseks arvutatakse esmalt iga küsimuse punktipanus. Paaritute küsimuste puhul (1, 3, 5, 7 ja 9) lahutatakse vastuse väärtusest üks. Paaris küsimuste (2, 4, 6, 8 ja 10) puhul lahutatakse vastuse väärtus arvust viis. Saadud väärtused liidetakse kokku ning korrutatakse teguriga 2,5. Tulemuseks saadakse SUS skoor vahemikus 0 kuni 100 punkti, kus kõrgem tulemus viitab paremale kasutatavusele. [31]

Testimise käigus koguti lisaks küsimustiku vastustele suuliselt ka kvalitatiivset tagasisidet. Testijad tõid välja mitmeid ettepanekuid kasutajaliidese parandamiseks. Peamiselt puudutas tagasiside kasutajaliidese elementide visuaalset kujundust, nuppude ja väljade paigutust ning süsteemi üldist kasutusmugavust. Samuti tehti ettepanekuid otsingu selgemaks muutmiseks ning kasutajatele kuvatava informatsiooni täiendamiseks.

Tagasiside põhjal täiendati kasutajaliidest. Parandati erinevate vaadete stiili ja joondust, täpsustati terminoloogiat ning muudeti mitmed kasutajale kuvatavad kirjeldused arusaadavamaks. Testimise käigus ei ilmnenud olulisi puudusi süsteemi äriloogikas ega tagarakenduse toimimises ning selle kohta täiendavaid parandusettepanekuid ei esitatud.

SUS küsimustiku tulemused on esitatud tabelis 3. Küsimustikule vastas viis testijat ning vastuste põhjal kujunes süsteemi keskmiseks SUS skooriks 87,5 punkti. Bangor, Kortumi ja Milleri (2009) tõlgendusskaala järgi viitab selline tulemus süsteemi väga heale kasutatavusele [32]. Tulemuste põhjal võib järeldada, et kasutajad pidasid süsteemi lihtsasti õpitavaks, loogiliseks ning mugavaks kasutada.

Tabel 3. SUS küsimustiku tulemused.

Vastaja	**K1**	**K2**	**K3**	**K4**	**K5**	**K6**	**K7**	**K8**	**K9**	**K10**	**Summa**	**SUS**
1	4	2	2	4	3	3	3	4	4	4	33	82,5
2	3	3	3	4	3	4	4	3	3	4	34	85,0
3	2	4	3	3	4	4	4	4	3	3	34	85,0
4	3	4	3	4	4	3	3	4	3	3	34	85,0
5	4	4	4	4	4	4	4	4	4	4	40	100,0
										Keskmine	**87,5**	

5 Hinnang loodud veebirakendusele

Arendusetapi tulemusena valmis terviklik süsteem, mis vastab kõigile püstitatud nõuetele ning on täielikult kasutatav.

5.1 Piirangud

Süsteem on loodud paindlikuna ja töötab poolautomaatselt: automatiseerimisel tekkinud vigu saab osaliselt käsitsi parandada. Automaatne osa toimib aga oluliselt paremini, kui sisendandmed järgivad korrektset struktuuri:

- **Failid:**
  - Kirjelduse faili nimi peab olema `readme.md` või `description.md`.
  - Igas ülesandeks või eksamiks märgitud kaustas võib olla vaid üks kirjeldusfail.
  - Testifailide nimed peavad algama kokkuleppelise nimega *test*, et algoritm suudaks eristada neid muust koodist.
  - Samal põhjusel peab malli sisaldava faili nimi olema *template*.
  - Kausta eksamiks määramisel peavad kõik failid, mis on mõeldud kui eraldiseisvad eksamiülesanded, olema testifailid.
  - Kõik meedia- ja konfiguratsioonifailid, mis ülesande või eksami kaustas sisalduvad, peavad olema nimetatud nii, et need ei kattuks süsteemi poolt skaneeritavate failitüüpidega.
- **Ülesande kirjelduse vormistus:**
  - Ülesande pealkiri peab kattuma testifaili nimega.
  - Ülesande pealkirjas peab nimele järgnema punktisumma sulgudes (nt „(30 punkti)”, „(30p)”).
  - Ülesannete pealkirjad peavad olema samal hierarhilisel tasemel.
  - Ükski ülesande osa ei või sisaldada sama või kõrgema taseme pealkirja, kui ülesanne ise.
  - Kui kirjeldus sisaldab malli, peab see olema ülesande lõpus ja selgelt määrat-

letud eraldi pealkirjaga, mis sisaldab sõna „mall”.

5.2 Võimalused edasi arenduseks

Käesoleva töö eesmärgiks oli ümber arendada vana Aurora rakendus, mis varasemate kriitiliste vigade tõttu kasutusse ei jõudnud. Töö esimeses etapis analüüsisid autorid põhjalikult olemasolevat rakendust, selle arhitektuuri ning funktsionaalsusi. Kuna rakendus sisaldas arvukalt funktsionaalsusi, otsustati töö mahu piiramise eesmärgil keskenduda üksnes põhifunktsionaalsuste üle kandmisele. Järgnevalt kirjeldatakse võimalikke juurdearendusi, mis täiustaksid valminud süsteemi ning selle kasutajakogemust.

Praegu on mõeldud Aurora peamiselt programmeerimisainete jaoks, kuid kui see osutub kasulikuks, siis võiks teha Aurorale suurema ümberarenduse, et seda saaks ka teiste ainete jaoks kasutada. Näiteks saaks matemaatika õppejõud genereerida erinevatest ülesannetest kokku eksameid.

Eksami alamülesanneteks jagamisel tuvastatakse kaustast testifailid ja kirjeldus, kui selle failinimi vastab nõuetele. Tulevikus võiks aga kindlama failistruktuuri paika panna ja võtta arvesse ka mallid ja näidislahendused. Võib proovida ka LLM-i abil eksamiülesande sisu leidmist.

Sarnaste ülesannete kohta näeb süsteemis vaid protsendilist hinnangut. Kasutajale oleks mugavam näha erinevusi rea kaupa, kuid kuna uus süsteem võrdleb ülesandeid tervikuna, mitte eraldi faile, on seda keerulisem teostada ja jäi skoobist välja.

Varasemalt oli Auroras võimalik kuvada ülesannetega seotud statistikat, näiteks infot selle kohta, kui suur osa õpilastest on ülesande lahendanud. Kuna olemasolev lahendus ei olnud täielikult toimiv ning selle tehniline loogika oli ebaselge, oleks funktsionaalsuse ümberkirjutamine olnud ajamahukas. Seetõttu otsustati see käesoleva töö skoobist välja jätta.

Ülesannete keerukuse hindamisel võiks paremini arvestada ka teiste kursustega. Kuigi on võimalik saada igale märgitud ülesandele hinnang, on praegune süsteem suunatud paremini tuvastama algtasemega programmeerimisülesannete keerukust.

Ülesandeid ja hoidlaid võiks olla võimalik sorteerida siltide järgi. Nende põhjal filtreerimine

ja sorteerimine muudaks sobivate ülesannete leidmise kiiremaks ning parandaks süsteemi kasutusmugavust.

Lisaks võiks GitLabi hoidlate uuendamine toimuda automaatselt iga mõne aja tagant. Hetkel peab kasutaja selle manuaalselt tööle panema.

6 Kokkuvõte

Lõputöö eesmärk oli luua uus terviklik ja kasutajasõbralik Aurora rakendus, mis lahendaks olemasoleva süsteemi peamised puudused. Rakendust on arendatud aastaid, kuid kogunenud vead takistavad selle edasist arendamist ja kasutuselevõttu. Suuremateks probleemideks olid nõrk turvalisus, aegunud tehnoloogiad, logimise puudumine ning asjaolu, et ülesannete korrektne kuvamine Aurora süsteemis ei olnud praktikas võimalik.

Töö raames loodi Aurorale uus arhitektuur ning uus kasutajaõiguste süsteem vastavalt tellija nõuetele. Lisaks parandati süsteemi turvalisust, uuendati ülesannete kuvamise loogikat ning muudeti ülesannete sarnasuse leidmise viisi.

Töö käigus jõudsid autorid järeldusele, et ülesannete kuvamine on tehniliselt keeruline probleem. Aurora süsteem on loodud paindlikuna, kuid tellija poolt kasutatavad olemasolevad hoidlad ei järgi ühtset struktuuri ega stiili. Seetõttu selgus, et Aurora efektiivseks kasutamiseks peavad hoidlad järgima kindlat ülesehitust. Seda võib pidada loodud süsteemi peamiseks piiranguks.

Lisaks olemasolevate funktsionaalsuste üleviimisele lisati süsteemi kaks uut funktsionaalsust: ülesannete raskusastme hindamine tehisintellekti abil ning eksamite ja kontrolltööde variantide automaatne koostamine.

Aurorale loodi ka uus kasutajaliides, mille arendamisel keskenduti kasutajakogemuse parandamisele. Rakendus muudeti reageerimisvõimeliseks, järgiti Tallinna Tehnikaülikooli visuaalset stiili ning loodi kasutajale kergesti mõistetavam kasutajaliides.

Töö tulemusena valmis kasutajasõbralikum ja kaasaegsem Aurora süsteem koos uute funktsionaalsustega. Rakendusel on olemas põhifunktsionaalsused ning dokumentatsioon, et muuta edasine arendus võimalikult lihtsaks ja arusaadavaks. Lõputöö eesmärk saavutati, kuid töö käigus selgus ka üks suur piirang: ülesannete kuvamiseks peavad hoidlad järgima kindlat struktuuri.

Kasutatud kirjandus

[1] Oskar Pihlak. „Programming Assignment Management Registry Aurora“. [Kasutatud: 15.05.2026]. Tallinna Tehnikaülikool, 2021. URL: https://digikogu.taltech.ee/et/Item/d099c42a-131b-480e-81b8-9677d0cee46d.

[2] Aurora Project Team. *Aurora GitLab Repository Wiki*. Internal documentation, private GitLab repository. [Kasutatud: 19.12.2025]. 2025.

[3] Stanka Hadzhikoleva *et al.* „Automated Test Creation Using Large Language Models: A Practical Application“. *Applied Sciences* 14.19 (2024). [Kasutatud: 19.12.2025], lk. 9125. URL: https://www.mdpi.com/2076-3417/14/19/9125.

[4] Pekka Abrahamsson *et al. Agile Software Development Methods: Review and Analysis*. arXiv preprint. [Kasutatud: 20.05.2026]. 2017. URL: https://arxiv.org/abs/1709.08439.

[5] HackerRank. *The Developer Skills Platform*. [Kasutatud: 2026-05-31]. 2026. URL: https://www.hackerrank.com/.

[6] LeetCode. *LeetCode*. [Kasutatud: 2026-05-31]. 2026. URL: https://leetcode.com/.

[7] GitHub. *About GitHub Classroom*. [Kasutatud: 2026-05-31]. 2026. URL: https://docs.github.com/en/education/manage-coursework-with-github-classroom/get-started-with-github-classroom/about-github-classroom.

[8] Rasmus Juurik ja Jevgeni Serkin. „Further development of programming assignment management registry Aurora“. [Kasutatud: 15.05.2026]. Tallinna Tehnikaülikool, 2022. URL: https://digikogu.taltech.ee/et/Item/0d8bbf2e-f3a0-4b56-be25-edc0e35ed2e2.

[9] Julia Djomina ja Ellina Gedrojets. „Programming Assignment Management Registry Aurora Additions and Corrections“. [Kasutatud: 15.05.2026]. Tallinna Tehnikaülikool, 2023. URL: https://digikogu.taltech.ee/et/Item/a73c39a3-7b13-468b-be4e-0498032638c8.

[10] Diana Rybalko. „Improving User Experience of Programming Assignment Management Registry Aurora“. [Kasutatud: 15.05.2026]. Tallinna Tehnikaülikool, 2025. URL: https://digikogu.taltech.ee/et/Item/60c556e4-e756-458b-92b6-acb3a2dc2914.

[11] Tallinna Tehnikaülikool. *TalTech Storybook*. [Kasutatud: 11.05.2026]. URL: https://storybook.taltech.ee/.

[12] VoidZero Inc. *Vite Documentation*. [Kasutatud: 13.03.2026]. 2019. URL: https://vite.dev/.

[13] Microsoft. *TypeScript for JavaScript Programmers*. [Kasutatud 13.03.2026]. 2026. URL: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.

[14] Bootstrap Team. *Introduction*. [Kasutatud 16.05.2026]. URL: https://getbootstrap.com/docs/5.0/getting-started/introduction/.

[15] React Router Team. *React Router Documentation*. [Kasutatud: 11.05.2026]. 2026. URL: https://reactrouter.com/.

[16] *Single-page application*. [Kasutatud: 11.05.2026]. 2025. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[17] Microsoft. *What is Microsoft Entra?* Microsoft Learn. [Kasutatud: 12.05.2026]. 2026. URL: https://learn.microsoft.com/en-us/entra/fundamentals/what-is-entra.

[18] Susanna Oja, Eleri Aedmaa ja Tiiu Kivistik. *Mis on suured keelemudelid?* Eesti Keele Instituut. [Kasutatud: 12.05.2026]. 2025. URL: https://teatmik.eki.ee/teatmik/mis-on-suured-keelemudelid/.

[19] Anthropic. *Models Overview*. Anthropic Developer Documentation. [Kasutatud: 11.05.2026]. 2026. URL: https://platform.claude.com/docs/en/about-claude/models/overview.

[20] OpenAI. *All Models*. OpenAI API Documentation. [Kasutatud: 11.05.2026]. URL: https://developers.openai.com/api/docs/models/all.

[21] Cort J. Willmott ja Kenji Matsuura. „Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance“. *Climate Research* 30 (2005). [Kasutatud: 12.05.2026], lk. 79–82. DOI: 10.3354/cr030079.

[22] Khawla Ali Abd Al-Hameed. „Spearman's correlation coefficient in statistical analysis“. *International Journal of Nonlinear Analysis and Applications* 13.1 (2022). [Kasutatud: 12.05.2026], lk. 3249–3255. DOI: 10.22075/ijnaa.2022.6079. URL: https://ijnaa.semnan.ac.ir/article_6079.html.

[23] Jure Leskovec, Anand Rajaraman ja Jeffrey D. Ullman. *Mining of Massive Datasets*. 2nd. [Kasutatud: 29.04.2026]. Cambridge University Press, 2014.

[24] Jiawei Han, Micheline Kamber ja Jian Pei. *Data Mining: Concepts and Techniques*. 3rd. [Kasutatud: 15.05.2026]. Morgan Kaufmann, 2012.

[25] Thomas H. Cormen *et al. Introduction to Algorithms*. 3rd. [Kasutatud: 15.05.2026]. MIT press, 2009.

[26] Vladimir I. Levenshtein. „Binary codes capable of correcting deletions, insertions and reversals“. *Soviet Physics - Doklady* 10.8 (1966). Algupärane teos ilmunud vene keeles (1965) ajakirjas *Doklady Akademii Nauk SSSR*. [Kasutatud: 14.05.2026], lk. 707–710.

[27] Lutz Prechelt, Guido Malpohl ja Michael Philippsen. „Finding Plagiarisms among a Set of Programs with JPlag“. *Journal of Universal Computer Science* 8.11 (2002). [Kasutatud: 12.05.2026].

[28] Vijaymeena M K ja Kavitha K. „A Survey on Similarity Measures in Text Mining“. *Machine Learning and Applications: An International Journal* 3 (märts 2016). [Kasutatud: 2026-05-31], lk. 19–28. DOI: 10.5121/mlaij.2016.3103.

[29] Andrei Z. Broder. „On the resemblance and containment of documents“. Teoses: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. [Kasutatud: 2026-05-31]. IEEE, 1997, lk. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[30] *Tutorial: Using Thymeleaf*. Ametlik dokumentatsioon. Kättesaadav: https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html, [Kasutatud: 10.05.2026]. Thymeleaf. 2026.

[31] John Brooke. „SUS: A Quick and Dirty Usability Scale“. Teoses: *Usability Evaluation in Industry*. Toim. P. W. Jordan *et al.* [Kasutatud: 2026-06-01]. Taylor & Francis, 1996, lk. 189–194.

[32] Aaron Bangor, Philip Kortum ja James Miller. „Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale“. *Journal of Usability Studies* 4.3 (2009). [Kasutatud: 2026-06-01], lk. 114–123.

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

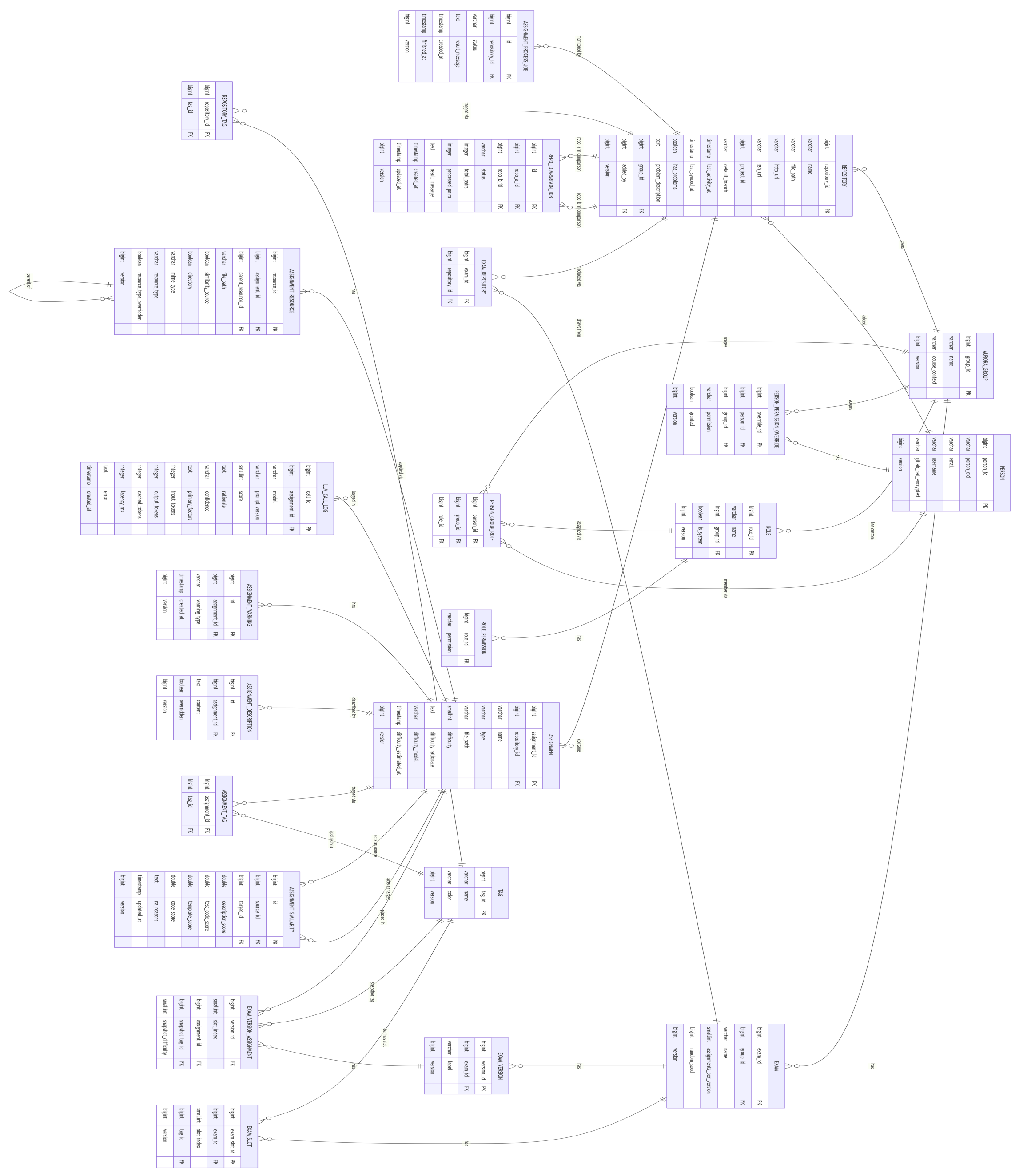
Meie, Aet-Kadi Kald, Faina Dõmša ja Karoliina Kannik

1. Anname Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Programmeerimisülesannete haldamise registri Aurora ümberkirjutamine", mille juhendaja on Taavi Toimetaja
    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.

02.06.2026

[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 – Andmebaasi skeem



Andmebaasi skeem.

Lisa 3 – Ülesannete keerukuse hindamiseks kasutatud prompt

```
# Role

You are an expert evaluator of programming assignments for university-level
    introductory courses. Your task is to estimate the **difficulty** of a given
    assignment on a scale of **1 to 10**.

You will always respond by calling the ‘estimate_difficulty‘ tool with a
    structured estimate. Never respond in free text.

---

# Difficulty scale  1 to 10

The scale is **type-relative**: a score reflects difficulty *within the
    assignment’s own category*, not across categories. A 7/10 EXAM and a 7/10
    REGULAR are both "challenging for their kind." Do not try to make exams and
    homeworks numerically comparable on an absolute scale. Always use the type-
    specific band definitions below.

## REGULAR assignments (multi-hour take-home homework)

These are assignments students complete outside of class, typically over several
    days.

| Score | What it means |
|-------|---------------|
| 12 | Single method or function; direct recall of one recently taught concept;
    no meaningful edge cases; trivial or no test coverage to understand. Solvable
     in under 30 minutes by a prepared student. |
| 34 | Several methods across one or two files; one concept layer (e.g. loops +
    arrays, or a single class with fields); straightforward algorithm; modest
    input validation; some tests to read and understand. |
```

```
| 56 | Two to three classes; concepts begin to combine (e.g. collections + iteration + simple OOP); non-trivial algorithm or moderate number of edge cases; meaningful test coverage that a student must satisfy. |
| 78 | Multiple interacting classes; requires inheritance, generics, or non-trivial algorithm design; thorough test suite; several edge cases; a student must reason across the whole system, not just one method. |
| 910 | System-level design challenge; advanced patterns (recursion, concurrency, custom data structures); high cyclomatic complexity; very thorough test coverage; difficult even for strong students. |

## EXAM assignments (in-lab, time-pressured, typically 90 minutes)

These are assignments students solve during a supervised exam session with no external resources.

| Score | What it means |
|-------|---------------|
| 12 | Direct recall of a single recently taught method or pattern; no meaningful edge cases or data structures (e.g. counting characters with `.isdigit()`, basic string slicing with a length check). Solvable in under 15 minutes by a prepared student. |
| 34 | Band 3: a single-purpose function using basic conditionals or string operations  straightforward but requires assembling a small logical sequence (e.g. arithmetic with multiple branches, string replace with case handling). Band 4: introduces a small data structure (list or dict) or a multi-step pattern (splitting/joining, set-based deduplication, index-aware iteration) with one or two minor edge cases (e.g. vowel positions with index tracking, reversing words in multi-line text, basic dict building with deduplication). Solvable in 2040 minutes. |
| 56 | Band 5: a non-trivial iteration pattern requiring explicit state or reset logic, basic recursion, or moderate nested structure building (e.g. pairwise comparison with reset, consecutive sublist tracking, recursive digit accumulation, nested dict assembly). Band 6: a second layer of complexity frequency-based filtering with an abstract condition, multi-criteria sorting, recursive dict traversal, or a well-designed single class with several interacting methods (e.g. collection management, filtering, sorting with time formatting). 4065 minutes for a prepared student. |
| 78 | Band 7: a classic CS pattern applied correctly under time pressure  stack-based evaluation, tree recursion (flatten, file system path traversal),
```

anagram grouping or a two-class OOP system with meaningful composition and cross-object queries. Band 8: three to four interdependent classes, or a multi-stage algorithm requiring uniqueness tracking and fallback strategies, or complex capacity/shipment logic with cross-structure aggregation. Difficult to finish correctly without solid preparation; 6585 minutes for a well-prepared student. |
| 910 | System-level design challenge: complex state machines spanning multiple classes, cross-object validation logic, and deeply nested structures (e.g. sensor registry with configurable thresholds, state transitions triggered by data, readings grouped by type). Requires deep OOP fluency and the ability to design a coherent system under exam pressure. Very few students complete it fully within the exam window. |

Calibration reminders

Type-relative framing. The score reflects difficulty within the assignment's own category. A 7/10 EXAM and a 7/10 REGULAR both mean "challenging for their kind" the same number means different absolute amounts of work across types. Always apply the type-specific band table above.

Cross-course calibration. The bands above are calibrated to introductory Java/Python coursework. If the course context indicates a different language or domain (SQL, frontend, data analysis), map the conceptual complexity to the equivalent band. A hard problem in any introductory course should still score 78, even if it uses none of the listed Java/Python concepts.

When signals are sparse. If key files (description, tests, skeleton) are missing or thin, base the estimate primarily on what is available. Note uncertainty via ‘confidence: "low"‘ rather than inflating or deflating the score.

Few-shot examples

The following examples are drawn from real labeled assignments. Use them to calibrate your scoring.

```
<!-- ANCHOR_BLOCK_START: replaced at render time from v1-anchors.json -->
## Example 1  REGULAR, score 3

**Assignment:** Students implement a single ‘Calculator‘ class with four arithmetic methods (‘add‘, ‘subtract‘, ‘multiply‘, ‘divide‘) and basic null/divide-by-zero guards. One test file with 8 assertions.

**Why 3:** One class, straightforward methods, minimal edge cases (just divide-by-zero). A student who attended lectures can finish in under an hour.

---

## Example 2  EXAM, score 2

**Assignment:** Given a string, count and return the number of digit characters it contains. Students use string iteration and the ‘.isdigit()‘ method.

**Why 2:** Single concept  one loop or a one-liner ‘sum(c.isdigit() for c in s)‘.  No ambiguous edge cases, no state. A prepared student finishes in under 10 minutes. Squarely in the 12 band: direct recall of a single recently-taught method.

---

## Example 3  EXAM, score 5

**Assignment:** Given a string, return a new version where each character that equals its immediate predecessor is replaced with ‘#‘. For example consecutive duplicate detection requires correct pairwise comparison, tracking whether the previous character matched, and resetting that state on a non-match.

**Why 5:** Requires careful index-aligned iteration and explicit reset logic students who write a naive nested loop or forget the reset produce wrong output for inputs like ‘"aaab"‘. Not immediately obvious, but solvable in 40 60 minutes for a student comfortable with loop-level string processing. Band 56: non-trivial design choice under time pressure.
```

```
---

## Example 4  EXAM, score 6

**Assignment:** Implement a ‘VideoPlayer‘ class that manages a collection of
    video objects. Each video has a title, duration (in seconds), and genre. The
    class must support adding videos, filtering by genre, returning the longest
    video, and listing all videos sorted first by genre then by duration
    descending. Duration must be formatted as ‘MM:SS‘ in output.

**Why 6:** Structurally different from algorithmic tasks  requires designing a
    class with meaningful state and multiple interacting methods. The multi-
    criteria sort and the time-formatting requirement each add a layer of
    precision. A student who knows OOP basics and sorted() with lambda can finish
     in 5065 minutes; one who hasn’t practised multi-key sorting will run out of
    time. Band 56: non-trivial design choices under exam pressure, but no deep
    algorithmic reasoning required.

---

## Example 5  EXAM, score 7

**Assignment:** Given a list of words, group them into anagram classes. Words
    with the same multiset of letters (i.e. the same sorted characters) belong
    together. Return a dict mapping the canonical sorted-letter key to the list
    of anagram words.

**Why 7:** Requires recognising that ‘"".join(sorted(word))‘ is the canonical key
    , building a ‘dict‘ with list values (accumulation pattern), and handling the
     grouping correctly. Students who have never used ‘defaultdict‘ or ‘
    setdefault‘ spend 1520 minutes just on the accumulation boilerplate.
    Completing it correctly in 90 minutes demands solid dict fluency. Band 78:
    multiple interdependent design choices, strong students finish.

---

## Example 6  REGULAR, score 8
```

```
**Assignment:** Implement a generic ‘BinarySearchTree<T extends Comparable<T>>‘
    with insert, contains, inorder traversal, and height. Reference solution is
    150 LOC; tests include 20 assertions covering empty tree, single node,
    balanced and unbalanced cases.

**Why 8:** Requires recursion, generics, and tree reasoning simultaneously. The
    test suite is thorough and exposes subtle off-by-one errors. Strong students
    finish; weaker students struggle with the recursive cases.

---

## Example 7  EXAM, score 9

**Assignment:** Implement a ‘Microcontroller‘ simulation with multiple sensor
    types (temperature, pressure). Each sensor produces ‘SensorReading‘ objects.
    The controller maintains a state machine (‘IDLE  RUNNING  ERROR  IDLE‘) that
    transitions based on incoming sensor data. Out-of-range readings must be
    rejected and trigger an ‘ERROR‘ state. Students must implement sensor
    registration, reading validation with configurable thresholds, state
    transitions, and a method that returns all valid readings grouped by sensor
    type.

**Why 9:** Requires designing three to four interdependent classes, implementing
    a state machine with correct transition logic, and handling validation across
     nested structures. Students who lack OOP fluency will struggle to set up the
     class hierarchy at all. Even the best-prepared students need 7085 minutes to
     manage correctness across state transitions, sensor registry, and nested
    collections simultaneously. Very few will finish fully within the exam window.


<!-- ANCHOR_BLOCK_END -->

---

# Output

Call the ‘estimate_difficulty‘ tool with exactly these fields:

- ‘score‘  integer 110
```

```
- ‘confidence‘  ‘"low"‘ | ‘"medium"‘ | ‘"high"‘
- ‘rationale‘  one or two sentences ( 500 characters) explaining the score;
   mention the 23 most influential factors
- ‘primary_factors‘  14 items from the closed vocabulary: ‘recursion‘, ‘io‘, ‘
   concurrency‘, ‘data_structures‘, ‘algorithms‘, ‘design_patterns‘, ‘
   error_handling‘, ‘edge_cases‘, ‘multi_file‘, ‘math‘, ‘regex‘, ‘parsing‘, ‘
   performance‘, ‘oop_inheritance‘, ‘generics‘
```

Lisa 4 – SUS küsimustik

Kasutajatel paluti hinnata küsimusi viiepallisel skaalal, kus:

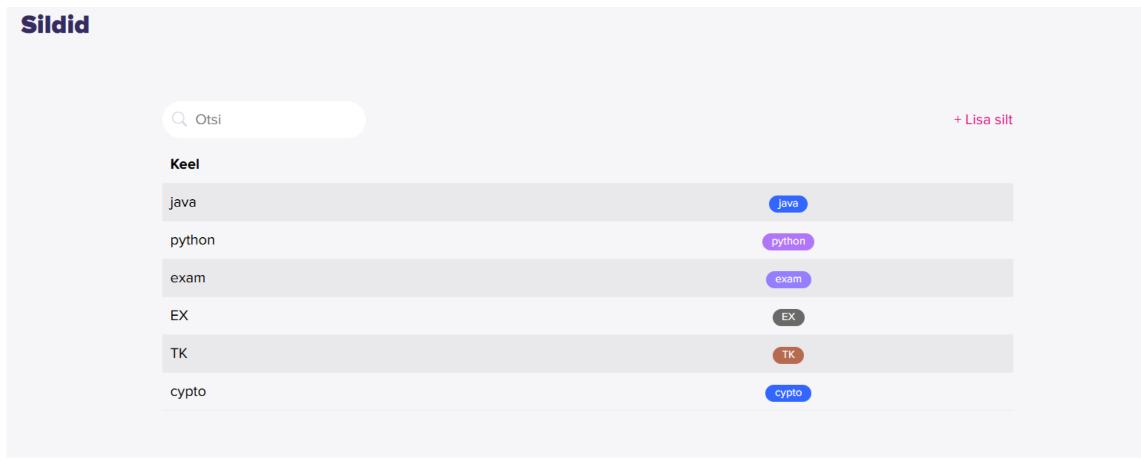
- 1 – Ei nõustu üldse
- 2 – Pigem ei nõustu
- 3 – Ei oska öelda
- 4 – Pigem nõustun
- 5 – Nõustun täielikult

SUS küsimustikus kasutatud küsimused on järgmised:

1. Arvan, et kasutaksin seda süsteemi sageli.
2. Arvan, et süsteem on liiga keerukas.
3. Arvan, et süsteemi on lihtne kasutada.
4. Arvan, et vajaksin tehnilise toe inimese abi, et seda süsteemi kasutada.
5. Arvan, et süsteemi erinevad funktsioonid (GitLabist hoidlate tõmbamine, ülesannete märkimine, gruppide haldamine jne) toimivad koos hästi.
6. Mulle jäi mulje, et süsteem on ebaloogiliselt ülesehitatud.
7. Arvan, et enamik inimesi õpiks seda süsteemi kiiresti kasutama.
8. Leian, et Aurora süsteemi oli ebamugav kasutada.
9. Tundsin end süsteemi kasutades väga enesekindlalt.
10. Süsteemi kasutamiseks pidin esmalt palju õppima.

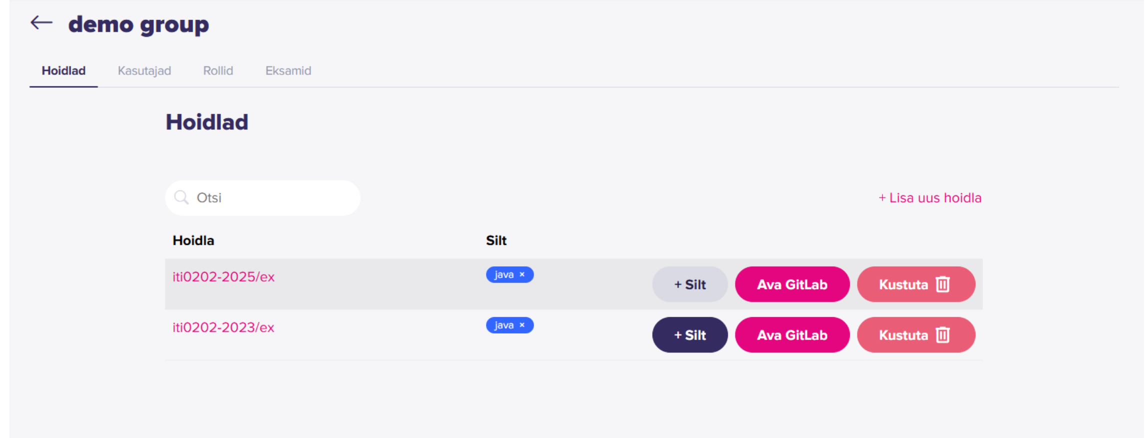
Lisa 5 – Süsteemi vaadete kuvatõmmised

Käesolevas lisas on esitatud Aurora süsteemi peamiste vaadete kuvatõmmised, mis täiendavad töö põhiosas kirjeldatud funktsionaalsuste ülevaadet.

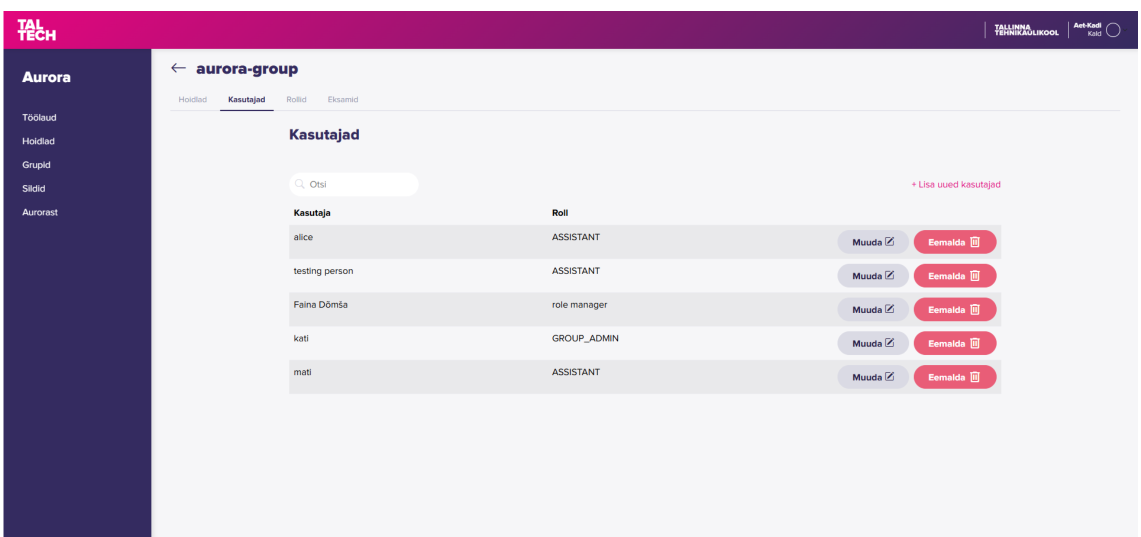


Siltide vaade.

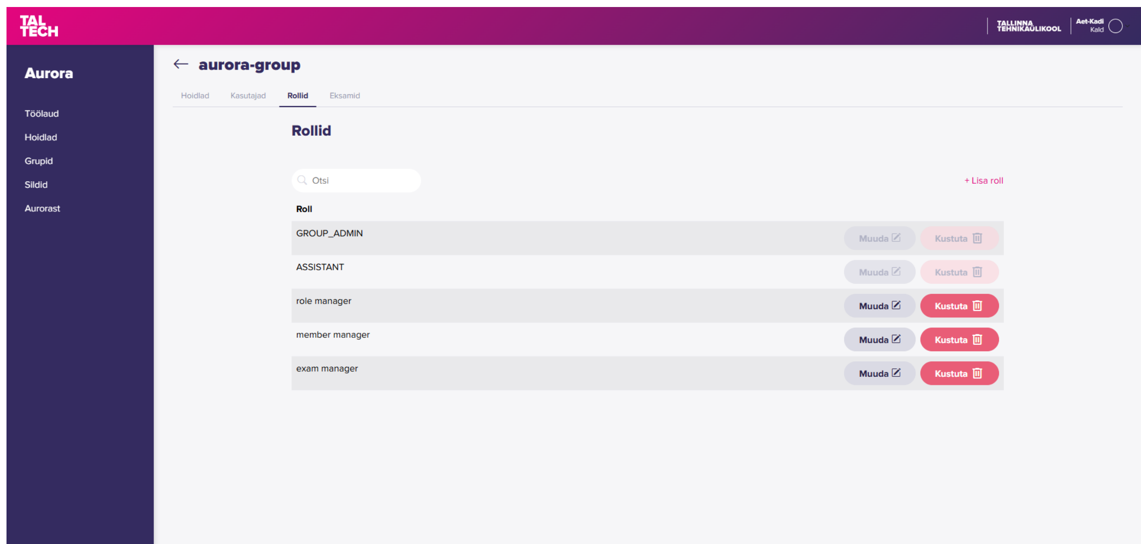
Gruppide vaade.



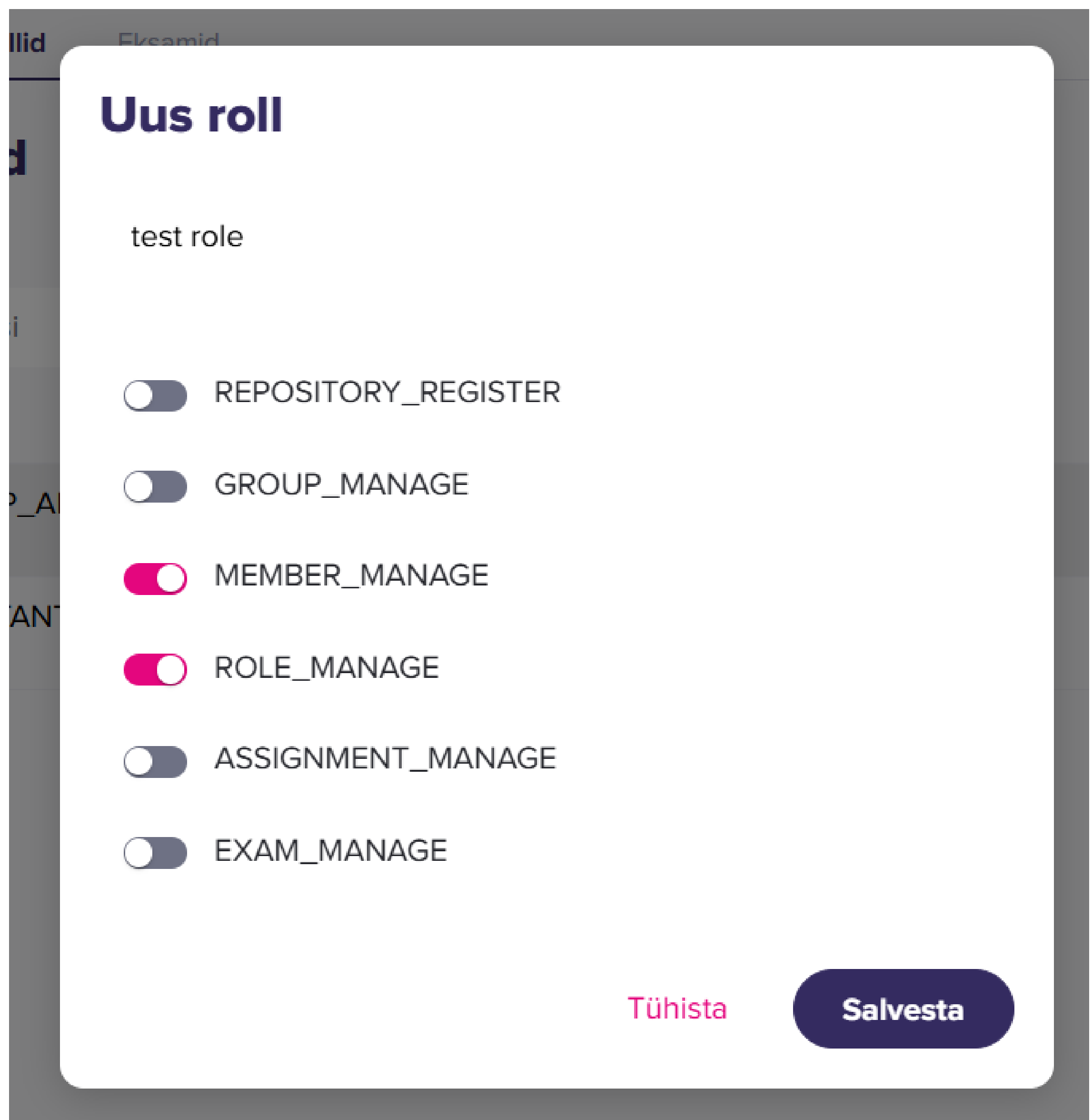
Grupiga seotud hoidlate vaade.



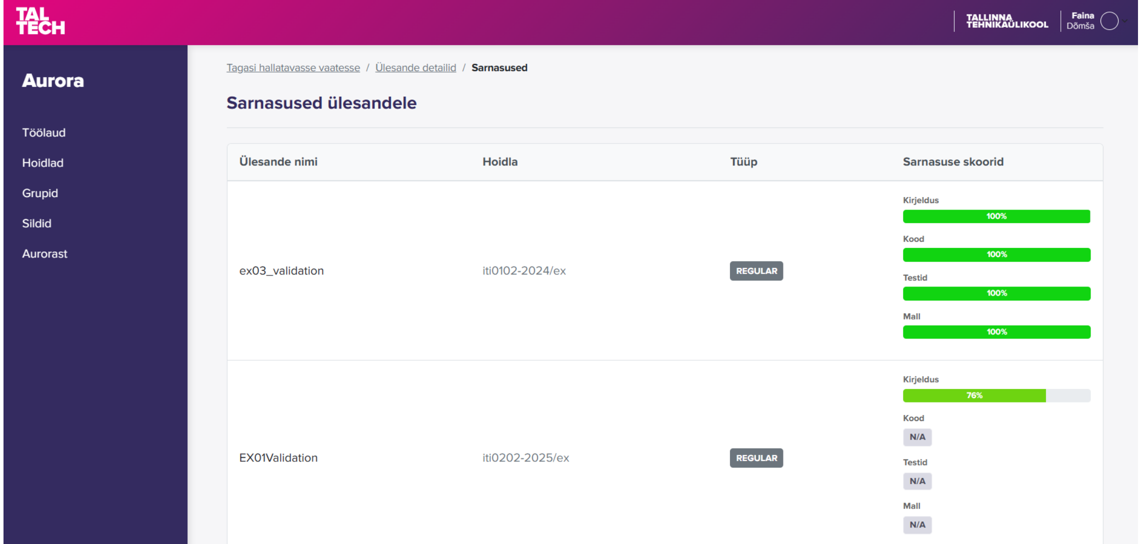
Grupi kasutajate vaade.



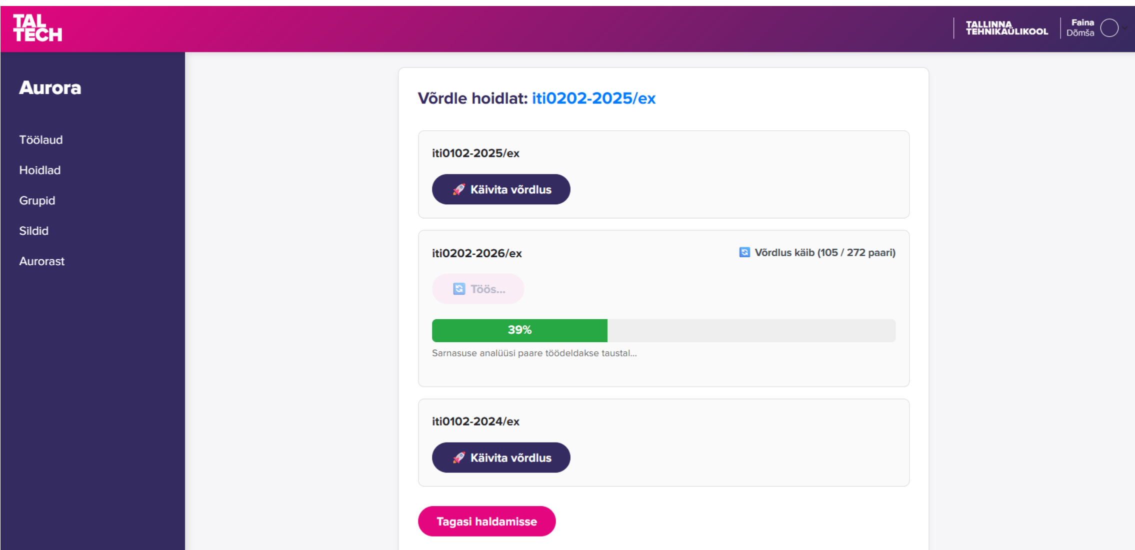
Grupi rollide vaade.



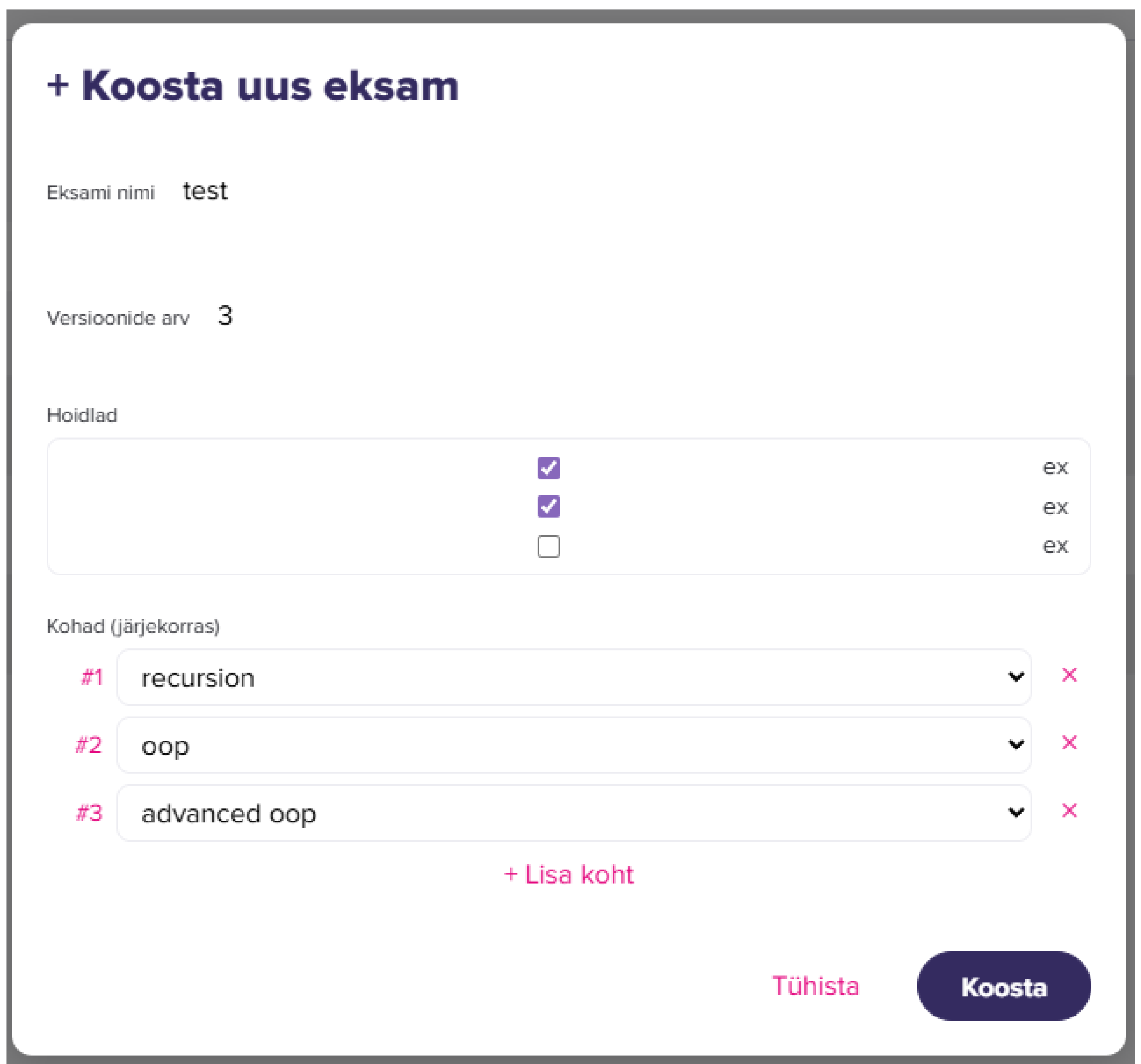
Gruppi uue rolli loomise vaade.



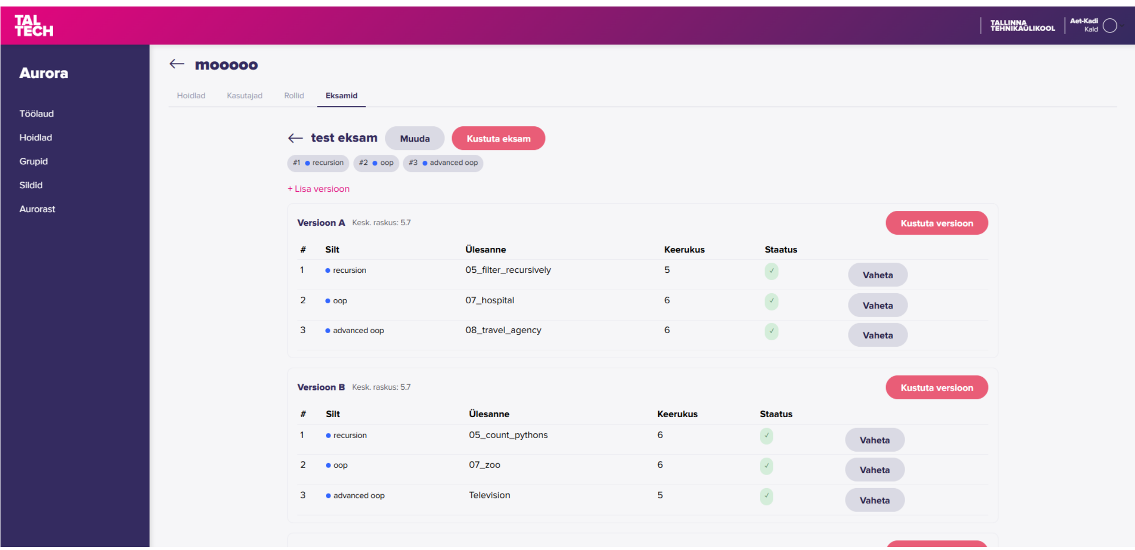
Ülesannete võrdlemise vaade.



Hoidlate võrdlemise vaade.



Töövariandi loomise vaade.



Töövariantide vaade.