TARTU ÜLIKOOL
Arvutiteaduste instituut
Informaatika õppekava

Karl Kevin Ruul

Energiahaldussüsteem pilvepõhise teenusena: arhitektuuriline ja arenduslik käsitlus

Bakalaureusetöö (9 EAP)

Juhendajad Tiit Sepp, Kirill Grjaznov

Tartu 2025

Energiahaldussüsteem pilvepõhise teenusena: arhitektuuriline ja arenduslik käsitlus

Lühikokkuvõte:

Töö üldiseks eesmärgiks on viia energiahaldussüsteem EEDA üle mikroteenustel põhinevale arhitektuurile ja tarkvara teenusena mudelile, parandades süsteemi hallatavust, vähendades majutuskulusid ning võimaldades kiiremat rakenduse kasutuselevõttu. Töö käigus loodi metoodika muudatuste teostamiseks, rakendati seda ning analüüsiti tulemusi. Uus rakendus täitis kõik seatud eesmärgid: säilitati kliendispetsiifilised konfiguratsioonid, kasutajaliidese funktsionaalsus, vähendati kulusid ning parandati arvutuskiirust keskmiselt üle 25% võrra.

Võtmesõnad:

Tarkvara teenusena, mikroteenused, energiahaldussüsteem.

CERCS: P175 Informaatika, süsteemiteooria

Energy Management System as a Cloud-Based Service: an Architectural and Developmental Approach

Abstract:

The main objective of the thesis is to transition the energy management system EEDA to a software as a service model using microservices-based architecture, improving system maintainability, reducing hosting costs, and enabling faster application deployment. A methodology for implementing the changes was developed, applied, and the results were analyzed. The new application met all set goals: client-specific configurations and user interface functionality were preserved, costs were reduced, and calculation speed improved by an average of over 25%.

Keywords:

Software as a service, microservices, energy management system.

CERCS: P175 Informatics, systems theory

Sisukord

Sissejuhatus 5
Lühendid ja mõisted 7
1. Taust 8
  1.1. Energiahaldussüsteem EEDA 8
    1.1.1. Rakenduse ülevaade 8
    1.1.2. Olemasoleva rakenduse probleemid 10
  1.2. Tehnoloogiliste aluste ülevaade 11
    1.2.1. Tarkvara teenusena 11
    1.2.2. Monoliitne arhitektuur 12
    1.2.3. Mikroteenustel põhinev arhitektuur 13
    1.2.4. Pilvtehnoloogia 14
    1.2.5. Virtualisatsioon ja konteinerdus 14
  1.3. Sarnaste funktsionaalsustega lahendused 17
    1.3.1. Virginia Tech'i energiahaldussüsteem 17
    1.3.2. Siemensi energiahaldussüsteem 17
2. Metoodika 19
  2.1. Süsteemiarhitektuur 19
  2.2. Kasutatavad tehnoloogiad 22
    2.2.1. Taustateenused 22
    2.2.2. Tuumsüsteemi teenused 23
    2.2.3. Kasutajaliides 23
    2.2.4. Tugiteenused 24
  2.3. Teenuste ülesehitus 26
    2.3.1. PV-ennustused 26
    2.3.2. Kalkulaator 27
    2.3.3. Optimeerija 30
3. Tulemused 32
  3.1. Arendatud mikroteenused 32
    3.1.1. Sisendite defineerimine 32
    3.1.2. Väljundite sõltuvuste haldamine 34
    3.1.3. Äriloogika klasside registrid 36
    3.1.4. Koodibaasi struktuur 37
  3.2. Vana ja uue süsteemi võrdlus 39

3.2.1. Sisulised muudatused 39
3.2.2. Kiirustestid 41
3.2.3. Majutuskulud 42
4. Võimalikud edasiarendused 44
Kokkuvõte 45
Viidatud kirjandus 47
Lisad 51
Lisa 1. Sisendplokkide näited 51
Litsents 52

Sissejuhatus

Soov elektrikuludelt säästa ja päikesepaneelide hinna langus on toonud elektri suurtootjate kõrvale üha enam tootvaid tarbijaid: majapidamisi, kes toodavad teatud määral ise elektrit. Arenguseire Keskuse andmetel oli aastal 2007 kuni 25 kW toomisvõimsusega tootvaid tarbijaid Eestis vaid üks, kuid aastaks 2022 on neid 9311. See moodustab 2% kõigist majapidamistest. Lisatakse, et kuivõrd taastuvenergiat on kulukas salvestada ning päikeseenergia näol on selle tootlikkus suurim ajal, mil majapidamine seda vajada ei pruugi, müüvad mitmed tootjad ülejääki elektrivõrku [1].

Tootvaks tarbijaks hakkamine nõuab arvestuslikke investeeringuid, mille tasuvus sõltub paljudest parameetritest, alustades elektrituru käitumisest ning lõpetades vahendite maksumuse ning nende kulumisega ajas [2]. Taastuvenergia investeeringu tasuvuse arvutamiseks ei ole ka standardiseeritud meetodit, sest näiteks päikesekiirguse või tuule hulk sõltub tootja asukohast. Seetõttu tekib tavainimesel vajadus spetsiaalse kalkulaatori järele, mis võimalikult paljusid mõjutavaid tegureid arvesse võtab.

Aktiivne elektritootmine lisab niigi keerulisse energiasüsteemi veel ühe kihi. Kui mittetootval tarbijal on ainukesed valikud elektri tarbimine ja selle eest paketipõhise hinna maksmine või elektri mitte tarbimine, siis aktiivsel tootjal on valikuid rohkem. Näiteks võib odava turuhinna kuid suure tootlikkusega hetkel otsustada võrgust tarbimise kasuks, salvestades enda toodetud energia akusse, et seda kõrgema hinnaga hetkel turule müüa või ise tarbida. Tulenevalt asjaolust, et Eesti elektriturg on 15-minutilise täpsusega, tuleb otsuseid teha palju. Selle jaoks kasutatakse energiahaldussüsteeme, mis arvutuslikult optimaalseid valikuid leiavad.

Andmeteadus- ja tarkvaraarendusettevõte STACC on loonud äriklientidele suunatud taastuvenergia tasuvuskalkulaatori ning energiahaldussüsteemi nimega EEDA [3]. Rakenduse tuum on koondpaneel, mida vastavalt klientide soovidele kohandatakse. Näiteks erinevad klienditi kalkulaatori sisendandmed ning arvutuste põhjal kuvatavad meetrikad ja joonised. Lisaks on osadele klientidele arendatud juurde ka veebipõhine rakendusliides (API) võimaldamaks integratsiooni nende süsteemidega. Igale kliendile luuakse selle jaoks koopia aluskoodist, mida kohendatakse ning seejärel eraldiseisvalt majutatakse.

Tulenevalt aluskoodi keerukusest ning tehnoloogiavalikutega kaasnevatest kitsendustest on rakenduse arendus muutunud vaevaliseks. Näiteks on kood monoliitse arhitektuuriga ning sealsed erinevad tükid tihedalt seotud, mis teeb sellest arusaamise raskeks. Uus klient peab praeguse tööprotsessi käigus ootama nädalaid või kuid kuniks rakendust arendatakse enne,

kui see tema jaoks kasutamiseks valmis on. Igale kliendile rakenduse eraldiseisvalt majutamine on ka kallis, sest arvutusressursse ei saa omavahel jagada.

Antud bakalaureusetöö on rakenduslik uurimus, mille eesmärk on muuta energiahaldussüsteem EEDA tarkvara teenusena kujul modulaarseks rakenduseks. See lähenemine lubab EEDA-t kasutada kui valmisteenust, mis on kliendi- või kasutajapõhiselt seadistatav. Tänu ühele kesksele süsteemile on majutuskulud väiksemad ning kliendil on võimalik rakendust koheselt kasutama hakata. Töö tulemusena eraldatakse rakenduse põhikomponendid mikroteenusteks, tehes seeläbi koodibaas hallatavamaks ning arendusprotsess kiiremaks. Kooditasemel luuakse ka struktuur seadistatavate sisend- ning väljundelementide lihtsaks muutmiseks ning nende lisamiseks. Töö eesmärgi saab lugeda saavutatuks kui:

1. kõikide olemasolevate klientide rakendusi on võimalik konfigureerida ning kasutada ühes tsentraalses rakenduses;
2. uue rakenduse kasutajaliideses on säilitatud kõik sisendid ning väljundid, kusjuures samade sisenditega arvutused annavad vana süsteemiga samu tulemusi;
3. uue rakenduse majutuskulud on väiksemad kui olemasolevate süsteemide kogukulu;
4. uue rakenduse arvutuskiirused ei ole aeglasemad olemasolevatest süsteemidest.

Töö on jaotatud nelja peatükki. Neist esimeses antakse ülevaade töö taustast. Kirjeldatakse detailsemalt energiahaldussüsteemi EEDA: selle seisu enne töös käsitletavate muudatuste läbiviimist ning olemasoleva lahenduse probleeme. Selgitatakse töös läbivalt käsitletavaid tehnoloogilisi aluseid, sealhulgas tarkvara teenusena mudelit, süsteemide arhitektuurimustreid ning pilvtehnoloogiaid. Antakse ka ülevaate seotud teadustöödest, mõtestades käesoleva töö vajadust ning konteksti. Teine peatükk analüüsib süsteemi nõudeid ning loob metoodilise põhja rakendusliku osa läbiviimiseks. Pannakse paika kogu süsteemi arhitektuur, kirjeldades loodavaid teenuseid ning kasutatavaid tehnoloogiaid. Sellele järgneb tulemuste analüüs, tuues välja esile kerkinud keerulisemad kohad koos kasutatud lahendustega ning valminud süsteemi võrdluse esialgse rakendusega. Viimane peatükk tõstatab ideid võimalikeks töö edasiarendusteks.

Lühendid ja mõisted

API (ingl *application programming interface*): rakendusliides

AWS: Amazon Web Services, Amazoni poolt pakutavad pilvtehnoloogia teenused

EEDA: STACC OÜ poolt arendatud energiahaldussüsteem

Energiahaldussüsteem: tarkvara energiakasutuse kontrollimiseks ning optimeerimiseks

ETAIS: Eesti Teadusarvutuste Infrastruktuur

IaaS (ingl *infrastructure as a service)*: taristu teenusena

Juurutamine (ingl *deployment*): tarkvara või teenuse kättesaadavaks tegemise protsess

Mastaabisääst (ingl *economies of scale*): kokkuhoid kliendi või ühiku pealt nende arvu kasvades

PV (ingl *photovoltaics*): valgusest elektri tootmine

SaaS (ingl *software as a service*): tarkvara teenusena

Skaleerimine (ingl *scaling*): arvutusressursside suurendamine või vähendamine vastamaks rakenduse või teenuse vajadustele

Taastuvenergia: looduslikest ressurssidest nagu päike, tuul või vesi tulenev energia, mis ei saa aja jooksul otsa

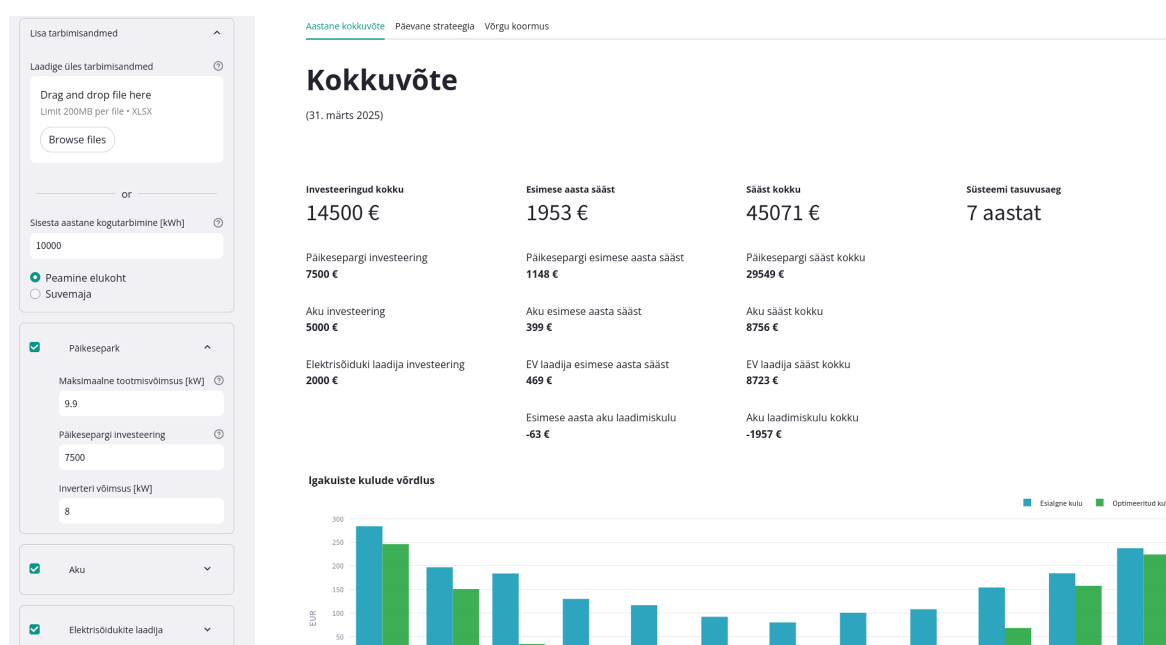
Taristu (ingl *infrastructure*): rakendust toetav riistvara ning tehnoloogia, näiteks server, võrguteenus

1. Taust

Taustapeatükk selgitab töö laiemat konteksti. Esmalt käsitletakse energiahaldussüsteemi EEDA, kirjeldades selle tööpõhimõtteid ning tuues esile olemasoleva süsteemi puudujäägid. Samuti määratletakse peamised probleemid, millele töö käigus lahendust leitakse. Seejärel selgitatakse tehnoloogilisi aluseid, sealhulgas olulisemaid mõisteid ja termineid, mida töös edaspidi kasutatakse. Viimaks antakse ülevaade olemasolevatest sarnastest lahendustest ning nende puudustest antud kontekstis.

1.1. Energiahaldussüsteem EEDA

Energiahaldussüsteem EEDA võimaldab kasutajal arvutada päikeseenergia investeeringu tasuvust, visualiseerida selle elutsüklit ning leida optimaalseim strateegia aku kasutamiseks. Selleks on kasutajal võimalik veebipõhisel töölaual määrata kindlate sisendite väärtused, mille põhjal arvutatakse vastavad mõõdustikud (ingl *metrics*), luuakse joonised ning tabelid. Näide töölauast on toodud joonisel 1.



Joonis 1. Energiahaldusplatvormi EEDA töölaud.

1.1.1. Rakenduse ülevaade

Sisendid jaotuvad kuude plokki: elektritarbimise-, päikesepargi-, aku-, elektrisõiduki-, elektrituru- ning finantseerimise andmed. Klienditi erinevad täpsemad sisendid plokisiseselt. Näiteks võib tarbimisplokis olla ühel kliendil vaid aastase kogutarbimise väli, teisel aga failiväli, mis võimaldab tunniajase täpsusega eelmise aasta andmete üles laadimist. Üldise

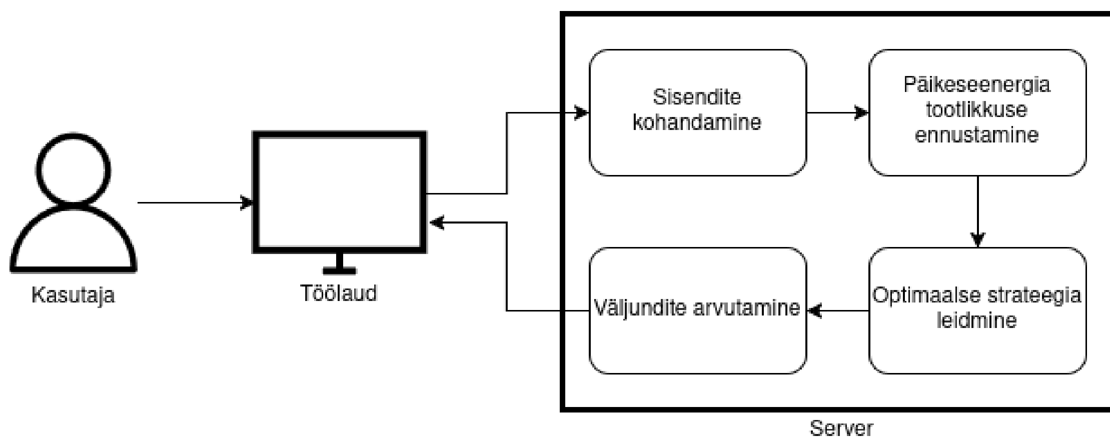
tööloogikana kohandatakse sisendeid sobitumaks osaliselt standardiseeritud andmestikuga, mille peal edasisi arvutusi läbi viiakse. Vajadusel laetakse sisenditele juurde ka väliseid andmed, näiteks elektrihindu. Päikesepargi sisendite põhjal luuakse asukohapõhised päikeseenergia tootlikkuse ennustused.

Sisendite, nendest tuletatud- ning juurde laetud andmete põhjal leitakse optimaalne strateegia aku kasutamiseks. Strateegia on oma olemuselt järjestatud tegevuste loend, mis peaks tagama suurima rahalise võidu. Tegevusi on kokku kuusteist ning nendeks võivad teiste seas olla näiteks (a) päikeseenergia kasutamine tarbimise katteks ja ülejäägi müümine turule, (b) aku laadimine päikeseenergiast ning (c) aku laadimine elektrivõrgust. Strateegia koostatakse üheks aastaks tunniajase täpsusega. Koos strateegiaga arvutatakse ka igatunnised energia liikumisvood. Need näitavad koguseliselt energia liikumist süsteemi eri komponentide vahel: näiteks kui palju energiat liikus päikesepaneelide tootlikkusest turule ning kui palju akust tarbimisse.

Optimaalse strateegia, energiavoogude ning sisendite põhjal arvutatakse viimaks välja mõõdustikud, luuakse joonised ning tabelid. Mõõdustikud koosnevad spetsiifilistest arvulistest näitajatest ehk mõõtudest planeeritava energiasüsteemi kohta. Mõõtudeks on näiteks investeeringu tasuvusaeg, süsteemi kogutulu või -kulu ja aku tsüklite arv. Kasutatavad joonised ning tabelid on peamiselt orienteeritud ajas muutuvate ehk kuude- ning aastatepõhiste näitajate ilustreerimiseks.

Rakenduse peamine sihtrühm on taastuvenergiaga tegelevad ettevõtted – päikesepaneelide paigaldajad, koduse elektritarbimise haldusplatvormid – kes vahendavad rakendust ja selle tulemusi kavandamaks oma klientide investeeringuid või optimeerides nende energiakasutust. Tulenevalt iga ettevõtte spetsiifilistest vajadustest on rakenduse aluskoodi kohandatud sisendite ja väljundite muutmiseks. Iga kliendi kohandatud rakendus on majutatud eraldiseisvalt.

Rakendus on kirjutatud Pythoni programmeerimiskeeles, toetudes suuresti Streamlit [4] raamistikule. Rakendus on oma ülesehituselt staatiline kalkulaator, mistõttu puudub vajadus kõrvaliste teenuste järele. Näiteks ei ole kasutusel andmebaasi. Toetavad funktsionaalsused nagu päikeseenergia tootlikkuse ennustamine ning optimaalse strateegia leidmine on realiseeritud otse rakenduse lähtekoodis. Rakenduse töövoog on kujutatud joonisel 2. Rakendus toimib sünkroonselt ehk päringuid teenindatakse ükshaaval ning ühtegi töövoo osa ei käidelda paralleelselt.



Joonis 2. Energiahaldussüsteemi EEDA töövoog.

1.1.2. Olemasoleva rakenduse probleemid

Tulenevalt klientide järjest keerulisematest nõuetest ning süsteemi praegusest ülesehitusest on arendusprotsess muutunud vaevarikkaks ja veaohtlikuks. Järgnevalt on välja toodud praeguse lahenduse probleemid ning kitsaskohad.

Streamlit raamistik on mõeldud veebipõhiste koondpaneelide koostamiseks. Kuigi Streamlit võimaldab lihtsat ja kiiret arendust, seab see keerulisemate nõuete korral teatud piiranguid. Raamistik värskendab iga muutuse, näiteks nupuvajutuse korral terve lehekülje [5], mis võib rakenduse aeglaseks muuta ning kulutab üleliigselt arvutusressursse. Pideva laadimise tõttu esineb praktikas suuremate andmemahtude korral mäluprobleemide tõttu ka lehekülje täielikku hangumist. Kuigi saadaval on osaline kasutajaliidese muutmisvõimekus, on see siiski raamistiku poolt piiratud ning lehekülje elementidele oma disaini rakendamiseks tuleb kasutada mittetriviaalseid lahendusi. Lisaks puudub raamistikul sisseehitatud või kergesti kasutusele võetav võimekus kasutajate halduseks[1], mida tarkvara teenusena kujul rakendusel vaja läheb.

Igale kliendile luuakse uus koodihoidla, mille aluseks kasutatakse rakenduse algelist versiooni ning mida hakatakse muutma ja täiendama vastavalt kliendi vajadustele. See aga tähendab, et arendajad peavad samaaegselt haldama mitmeid koodihoidlaid, mis on ajakulukas ning sisaldab tihti ka samasuguse, monotoonse töö tegemist. Keerulisem on turvauuenduste elluviimine ning avastatud vigade parandamine, mida tuleb teha igas projektis

[1] 2025. aasta veebruaris lisati funktsionaalsus kasutajate autentimiseks, kuid kasutajate haldus on töö kirjutamise hetkel veel eksperimentaalfaasis: https://docs.streamlit.io/develop/concepts/connections/authentication

eraldi. Koodihoidlad ei ole omavahel sünkroniseeritud ning seetõttu on uute võimekuste lisamine kliendipõhine.

Iga kliendi projekt on majutatud eraldiseisvalt. Ühe projekti jaoks on kasutatud Streamlit raamistiku poolt pakutavat Community Cloud teenust [6], ülejäänud projektid on majutatud kasutades Amazon Web Service't (AWS), kuhu on igale kliendile loodud oma konto. Lisaks mitmetele koodihoidlatele tuleb seetõttu hallata ka mitmeid kontosid erinevate teenusepakkujate juures. Projektide eraldatuse tõttu ei saa need kasutada ühiseid pilveressursse ning majutuskulud on kõrgemad kui need võiksid olla ühtse süsteemi korral, varieerudes klienditi 100 ning 250 euro vahel kuus.

Rakenduse kiire arenduse ning testide puudulikkuse tõttu on sinna sisse jäänud mitmeid pisivigu, mis tulevad tihtipeale välja kasutajate tagasisidest. See tähendab kliendile aga ebameeldivat kasutajakogemust, sest vead takistavad rakenduse kasutamist. Tulenevalt asjaolust, et kõik uued projektid kloonitakse baasprojektist, kanduvad vead kergesti ka üle.

1.2. Tehnoloogiliste aluste ülevaade

Tehnoloogiliste aluste ülevaate alapeatükk selgitab edasises töös läbivalt kasutuses olevaid termineid ning nende taga peituvaid tehnoloogilisi lahendusi. Vaatluse all on tarkvara teenusena mudel, monoliitne- ja mikroteenustel põhinev arhitektuur, virtualisatsioon ning konteinerdus.

1.2.1. Tarkvara teenusena

Tarkvara teenusena (SaaS) mudel kujutab endast pilvepõhist, kasutajate vahel jagatud ressurssidega veebirakendust, millele saadakse ligipääs läbi regulaarsete maksetega tellimuse või kasutuspõhise hinnastamisega [7, 8]. See on vastand traditsioonilisele tarkvarale, mis ostetakse välja ühekordse maksega ning kasutatakse seejärel kohalikus keskkonnas, teistest kasutajatest sõltumatult [7]. Üheks näiteks tarkvara teenusena pakkumisest on Microsoft 365, mis on alates 2017. aastast tellimuspõhine alternatiiv Microsoft Office programmide ostmisele [9].

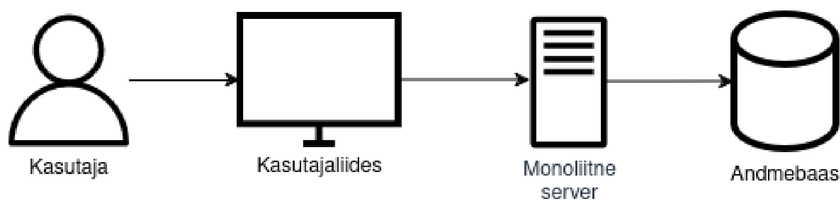
Ouh ja Kok Siew Gani sõnul soovivad tarkvara teenusena pakkujad suurendada tootlikkust läbi mastaabisäästu, pakkudes teenust rohkematele kasutajatele, ilma, et kaasnevad kulud oluliselt suureneksid. Nad väidavad, et selle saavutamiseks tuleb tarkvara majutada kasutades ühiseid arvutuslikke ressursse, kaasa arvatud ühist andmebaasi. Veel lisatakse, et

teenusepakkuja peab võimaldama piisava personaliseerimisvõimekuse, et katta erinevate kasutajate vajadused [8].

Shapouri *et al.* läbi viidud uuringust selgub, et tarkvara teenusena kasutavad suurema tõenäosusega nii väiksema käibe kui ka väiksema infotehnoloogia (IT) osakonnaga ettevõtted. Argumenteeritakse, et väiksema käibega ettevõtetele tulevad kasuks madalamad IT kulutused, mis saavutatakse hoidudes rakenduse välja ostmisest, ning võimalus igal hetkel kulutustest loobuda. Väiksema IT osakonnaga ettevõtetes ei ole nii suurt tehnoloogilist oskusteavet kui suurema IT töötajaskonnaga ettevõtetes, mistõttu eelistatakse teenusepakkuja poolt hallatud tarkvara teenusena, väidavad Shapouri *et al* [10].

1.2.2. Monoliitne arhitektuur

Monoliitne arhitektuur on tarkvara ülesehituse meetod, kus kogu rakendus ning selle tööks vajalik on koos ühes komplektis. Kapikul *et al.* selgituse põhjal koosneb tüüpiline rakendus kasutajavaatest, serveri komponendist ja andmebaasist ning monoliidiks saab pidada serverit, mis haldab kogu rakenduse suhtlust kasutajavaate ning andmebaasi vahel [11]. Sellist monoliitset arhitektuuri on kujutatud joonisel 3. Newman käsitleb monoliiti kui juurutusüksust (ingl *deployment unit*), ehk rakendust, mida tuleb juurutada tervikuna. Ta eristab lisaks ka hajusmonoliiti, kus teenused on sisemiselt loogiliselt eraldatud mooduliteks [12].



Joonis 3. Monoliitse arhitektuuriga rakenduse ülesehitus.

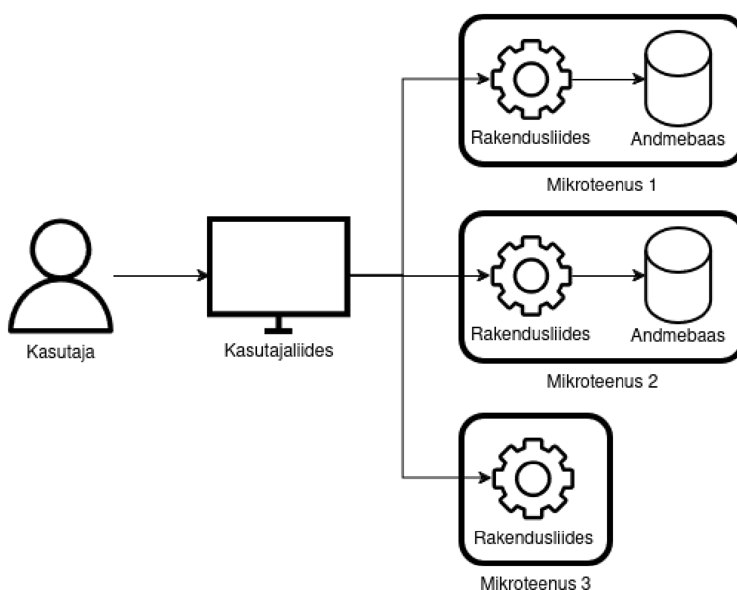
Gos ja Zabierowski poolt läbi viidud eksperimendist järeldati, et monoliitse arhitektuuri kaks eelist on arendamise lihtsus ning lihtne juurutamine. Võrreldes mikroteenustel põhineva arhitektuuriga leiti, et monoliitse rakenduse arenduses ei teki nii palju probleeme integratsioonide, ühenduste ning konfiguratsioonidega. Lisaks märgitakse, et eksperimendi käigus arendatud rakendus on madala koormuse korral tõhusam. Kuigi juurutamise lihtsust

eraldi pikemalt ei käsitleta, on ilmne, et üht terviklikku tarkvarakomplekti on mitmest eraldiseisvast teenusest lihtsam juurutada [13].

Nagu tuuakse välja mitmes uuringus [11, 14, 15], on peamine monoliitse arhitektuuri puudujääk selle halb skaleeruvus. Kui rakenduse mõne komponendi kasutus või ressursivajadus suureneb, tuleb monoliitse arhitektuuri korral skaleerida kogu rakendust. See tähendab aga omakorda ebaefektiivsemat ressursikasutust tervikliku rakenduse peale. Veel tuuakse puudujääkidena välja keerulist koodibaasi haldust suurema projekti korral, sõltuvust ühest programmeerimiskeelest ning kindlatest raamistikest ja vajadust ka kõige väiksema muudatuse korral terve rakendus taaskäivitada [14].

1.2.3. Mikroteenustel põhinev arhitektuur

Pathak ja Singh tõlgendavad mikroteenuseid kui sõltumatuid, madala sidususega teenuseid, millest igaüks keskendub kindlale ärieesmärgile. Nad lisavad, et sõltumatus tuleneb iga teenuse eraldiseisvast arendusest, majutusest ning skaleerimisest [16]. Mikroteenustel põhinev arhitektuur eristab erinevalt monoliitsest arhitektuurist väga selgelt eri komponente ehk teenuseid, mis tervikuna moodustavad ühe rakenduse. Igal teenusel võib olla vastavalt vajadusele oma andmebaas ning ainukeseks ühenduspunktiks on API, millega suheldakse teiste teenuste ning kasutajaliidesega [11]. Mikroteenustel põhinevat arhitektuuri illustreerib joonis 4. Mikroteenuste kasutamine on muutunud väga populaarseks ning ennustatakse, et 90% tulevastest rakendustest arendatakse just mikroteenustel põhineva arhitektuuriga [17].



Joonis 4. Mikroteenustel põhineva arhitektuuriga rakenduse ülesehitus.

Gos ja Zabierowski eksperimendi põhjal on mikroteenustel põhineva arhitektuuriga rakendust monoliitsest rakendusest kergem hallata, sest kõik põhifunktsionaalsused on üksteisest eraldatud ning arendajal seeläbi lihtsam rakendusest aru saada [13]. Berry *et al.* katsed näitavad, et suure koormusega rakenduse puhul nõuab mikroteenuste skaleerimine vähem ressursse optimaalse jõudluse saavutamiseks, sest kogu rakenduse asemel saab juurde luua vaid kindlate teenuste isendeid [18]. Seega võib järeldada, et mikroteenused on sobilik lahendus keerukate või suure koormusega rakendustele.

Pathak ja Singh toovad mikroteenustel põhineva arhitektuuriga kaasnevateks väljakutseteks muuhulgas teenustevahelise suhtluse ja sõltuvuste haldamise, keerulisema süsteemi jälgitavuse ning andmete kooskõlas hoidmise [16]. Arvestades, et iga mikroteenus on justkui eraldiseisev väike rakendus, on sellise arhitektuuriga kaasnevad suuremad haldusvajadused ning -raskused igati ootuspärased.

1.2.4. Pilvtehnoloogia

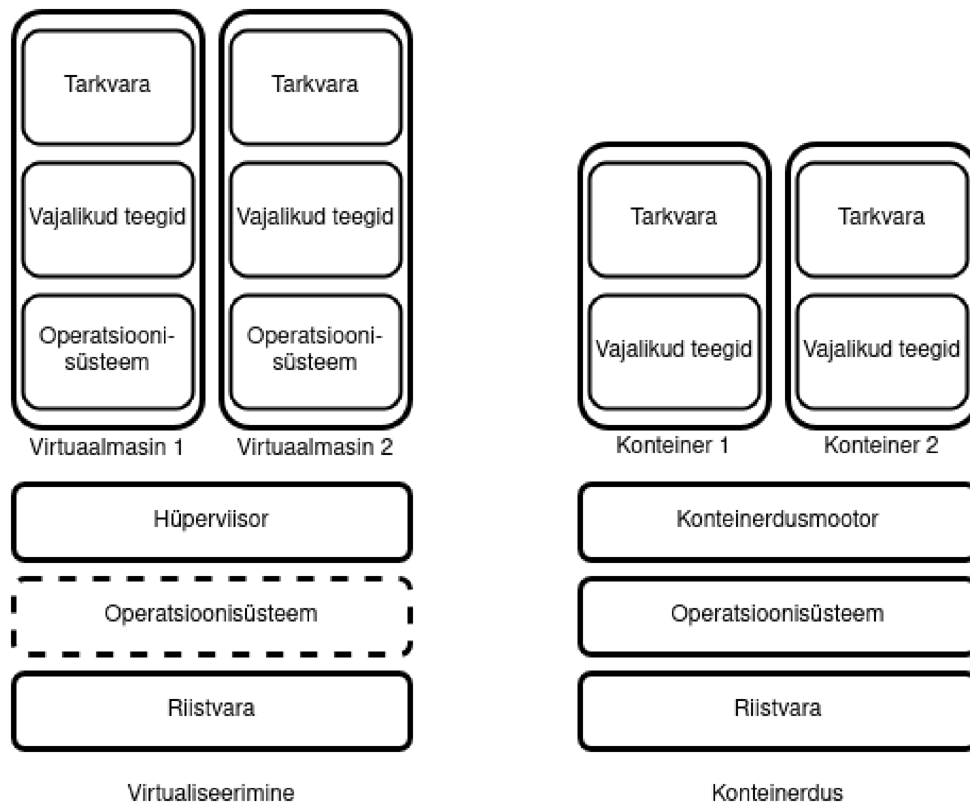
Pilvtehnoloogia (ingl *cloud computing*) on muutunud üheks populaarseimaks võtmesõnaks IT-sektoris. Google'i otsingusõnade statistikast leidub, et suur hüpe otsingute arvus leidis aset vahemikus 2008-2010, pärast mida on see olnud pidevalt kõrgel [19]. Pilvtehnoloogia lubab võrgupõhist ligipääsu jagatud arvutusressurssidele nagu näiteks serveritele, andmeladudele ning teenustele, mida on kasutajal võimalik konfigureerida, defineerib USA Riiklik Standardi- ja Tehnikainstituut [20]. Tänu pilvtehnologiale puudub IT-teenuste pakkujal kohustus enda riistvara ostmiseks ning selle ülalhoidmiseks, võimaldades seeläbi madalamaid kulusid ning suuremat paindlikkust.

Sharma uurimuse järgi pakuvad pilvtehnoloogiad peamiselt kolme eri tüüpi teenuseid, olenevalt nende ülesehitusest ja kasutustüübist: tarkvara teenusena (SaaS), taristu teenusena (IaaS) ning platvorm teenusena (PaaS). Tarkvara teenusena on lõppkasutajale suunatud veebirakendus koos kasutajaliidesega. Taristu teenusena pakub kasutajale teenusepakkuja poolt hallatavaid servereid, andmebaase või muid IT-taristu komponente, millele on võimalik süsteeme rajada. Platvorm teenusena on pilvepõhine arenduskeskkond tarkvara loomiseks [21].

1.2.5. Virtualisatsioon ja konteinerdus

Virtualisatsioon on abstraktsioonikiht, mis eraldab operatsioonisüsteemi riistvarast, võimaldades seeläbi mitmel virtuaalmasinal ehk -keskkonnal isoleeritult ühe ja sama füüsilise masina peal joosta [22]. Nanda ja Chiueh defineerivad virtualiseerimise kui tehnoloogia, mis

kombineerib või jagab arvutusressursse mitme töökeskkonna ülalhoidmiseks, kasutades selleks muuseas riist- ning tarkvara sektsioneerimist või agregeerimist [23]. Nende kirjelduse põhjal on riistvara ning virtuaalmasinate vahel käske edastav hüperviisor (vt joonis 5), mis omab täielikku kontrolli füüsilise masina üle ning suudab virtuaalmasinaid eraldatuna hoida. Virtualisatsioon võimaldab vähesema riistvara haldusega rohkemate tarkvarade jooksutamist ning loob turvalise keskkonna rakenduste testimiseks eri platvormidel [24].



Joonis 5. Virtualiseerimine ja konteinerdus kahe tarkvara käitamiseks.

Bentaleb *et al.* sõnul on konteinerdus (ingl *containerization*) kergekaaluline alternatiiv virtualisatsioonile [25]. Nende sõnul tuleneb see asjaolust, et riistvarakihi asemel abstraheeritakse läbi konteinerdusmootori operatsioonisüsteemi, kasutades virtuaalmasinate asemel operatsioonisüsteemita konteinereid (vt joonis 5). Konteinerite käivitamine on seetõttu võrreldes virtuaalmasinatega kiirem ning mitmete testide põhjal jõudlus parem.

Virtualisatsioon ning konteinerdus on ühtedeks tänapäevase pilvtehnoloogia alustaladeks [26]. Varasemalt kasutusel olnud eraldatud masinate operatiivkulud olid suured ning keskmiselt kasutati ära vaid kuni 30% olemasolevast jõudlusest, virtualiseerimine ja konteinerdus lubavad aga virtuaalmasinate või konteinerite seadistamist füüsiliste masinate

vahel selliselt, et ressursikasutus oleks võimalikult efektiivne [24]. See vähendab andmekeskuste energiatarbimist ning kulusid [27]. Lisaks kergenadavad need tehnoloogiad süsteemide skaleerimist ning kiirendavad juurutamist [28].

1.3. Sarnaste funktsionaalsustega lahendused

Mõistmaks antud bakalaureusetöö konteksti ning lahendatavaid uudseid probleeme, tuleb analüüsida olemasolevat teaduskirjandust sarnastest lahendustest. Analüüs käsitleb seotud tööde sisu, lahenduskäike ning nende puudujääke või mittesobivust käesoleva töö kontekstis. Vaatluse all on kaks välisülikoolide ning -tehnoloogiafirmade poolt arendatud energiahaldussüsteemi.

1.3.1. Virginia Tech'i energiahaldussüsteem

Virginia Tech teadlased on loonud mikroteenustel põhineva energiahaldussüsteemi tarkvara teenusena arhitektuurilahenduse [29]. Rakendus on mõeldud eeskätt tarkade hoonetega värkvõrgu (IoT) kaudu ühildumiseks, võimaldades hoone energiatarbimise jälgimist ning reguleerimist. Tuuakse välja lahendus kolmekihilise süsteemi jaoks: teenuskiht, rakendusliideste kiht ning baaskiht. Neist esimene on mõeldud peamiselt kasutajaliideste jaoks, viimane tugiteenuste (näiteks erinevate energiakomponentide võrgust leidmise ja nende juhtimise) jaoks ning teine eelneva kahe vaheliseks suhtluseks. Arhitektuurilahendus on põhjalikult läbi mõeldud ning detailne, kirjeldatud on isegi rakendusliidese otspunktid. Lisaks on läbi viidud jõudlustestid AWS-is ning tulemusi on võrreldud monoliitse rakendusega.

Küll aga puuduvad lahenduses mitmed tänapäevase energiahaldussüsteemi komponendid: elektritootmise jälgimine ja sellega arvestamine, automaatne energiatarbimise ning -tootmise reguleerimine ning elektrituru, sealhulgas elektrihinna arvestamine. Teisisõnu peitub komplekse rakenduse taga sisult lihtne süsteem, mis võimaldab andmete eraldamist ning seadmete manuaalset reguleerimist, kuid millel puudub reaalne haldusvõimekus. Lisaks ei ole antud lahenduses käsitletud kasutajaspetsiifilist seadistamist, mis EEDA kontekstis on üheks peamiseks väljakutseks ning ka müügiteguriks.

1.3.2. Siemensi energiahaldussüsteem

Siemens, koostöös Londoni avaliku sektori asutuste ning mitme maineka briti ülikooliga, on välja töötanud mikroteenustel ning hüpervõrkudel (ingl *hypernetworks*) baseeruva energiahaldussüsteemi lahenduse [30]. Töö fookuseks on võetud kindlad sotsiaalmajad Londonis, Greenwichis. Lahendus on oma olemuselt terviklik: käsitletud on nii tarbimist (sealhulgas eri komponendid nagu gaasiboiler ja elektriautode laadijad), tootmist läbi päikesepaneelide ning on loodud ka väliseid parameetreid (näiteks elektri- ning gaasihinnad

ja õhutemperatuur) arvesse võttev päev-ette ennustus- ja optimeerimismudel. Arhitektuuriliselt on tarkvaralahendus jaotatud nelja kihti: detailkontrolli kiht erinevate energiakomponentide reguleerimiseks, keskkontrolli kiht terve hoone reguleerimiseks, üldkontrolli kiht optimeerimiseks ning eraldi kiht kolmandate osapoolte andmete käsitlemiseks. Süsteemi konfigureerimine on võimalik kasutades hüpervõrke, mis on oma olemuselt keerulised matemaatilised struktuurid [31].

Kuigi rakendus on oma komponentide ja nendega kaasnevate võimekuste poolest paljulubav, on selle kasutusjuht praktikas kitsas. Täpsemalt on süsteem optimeeritud töötama kolme spetsiifilise hoone jaoks, mistõttu on strateegia optimeerimist raske, lisaandmete puudumise korral potentsiaalselt võimatu laiendada kohandumaks teiste regioonide või tingimuste jaoks. Loodud on küll tugi energiasüsteemi konfigureerimiseks, kuid see piirdub komponentide ning nende spetsifikatsioonide kirjeldustega. Võrdluses EEDA-ga oleks justkui võimekus kindlate sisendite täitmiseks, kuid sisendite endite muutmine pole võimalik. Lisaks ei keskendu lahendus täiemahulisele kasutajale suunatud tarkvarale, käsitletakse vaid energiahaldussüsteemi tuuma ehk optimeerimise tagaliidest.

2. Metoodika

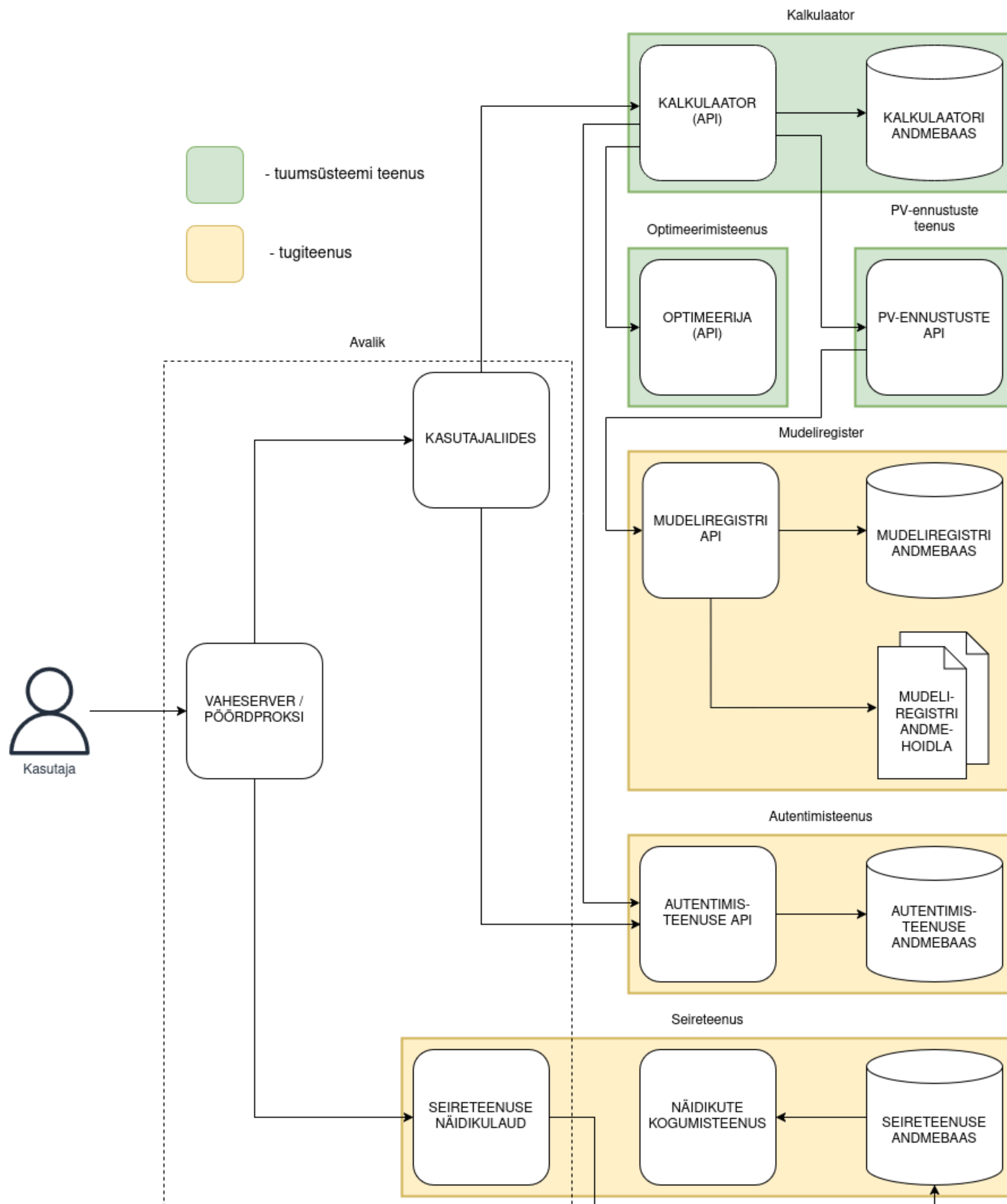
Käesoleva bakalaureusetöö raames viiakse energiahaldussüsteem EEDA üle tarkvara teenusena mudelile. Uus rakendus peab olema iga kliendi jaoks konfigureeritav: peab olema võimekus nii sisendite kui väljundite muutmiseks ning sobivate puudumise korral peab nende lisamine koodis võimalikult lihtne olema. Seeläbi kaob vajadus iga kliendi jaoks eraldiseisva rakenduse loomiseks ning selle majutamiseks, mis eelduste kohaselt langetab kulusid [8]. Lisaks võib eeldada, et mitmete koodibaaside asemel ühe haldamine tõstab üleüldiselt teenuse kvaliteeti, sest lihtsam on teha uuendusi ning neid ka testida.

Ajaga on EEDA muutunud üha keerulisemaks rakenduseks, kuhu on vaevarikas uut funktsionaalsust lisada ning kuhu on uuel arendajal raske sisse elada. Üheks probleemi põhjustajaks saab pidada olemasolevat monoliitset arhitektuuri, mis sisaldab endas suurel hulgal tihedalt seotud koodi. Selle parandamiseks luuakse uus süsteem mikroteenuste baasil, mis eraldab üksteisest suuremad komponendid. See muudab omakorda koodibaasi hallatavamaks [13].

Järgnevates alapeatükkides kirjeldatakse üleviimiseks vajalikke tegevusi, tuues välja ka detailsemad plaanid, joonised ning kasutatavad tehnoloogiad nende teostamiseks.

2.1. Süsteemiarhitektuur

Tulenevalt Streamliti puudujääkidest ning vajadusest dünaamilise rakenduse järele, eraldatakse üksteisest taga- ning kasutajaliides. Teisisõnu võetakse kasutajaliidese jaoks kasutusele selleks ettenähtud raamistik ning ehitatakse tagaliides üles viisil, mis võimaldaks kasutajaliidesel sellega suhelda. Tagaliides lüüakse omakorda lahku mikroteenusteks, mis omavahel läbi APIde suhelda saavad. Uue tuumsüsteemi moodustavad neli teenust: kasutajaliides, kalkulaator, päikeseenergia tootlikkuse ennustamine (edaspidi PV-ennustamine) ning optimaalse strateegia leidmine (edaspidi optimeerija). Loodava süsteemi arhitektuuri illustreerib joonis 6.



Joonis 6. Loodava süsteemi teenuste arhitektuur.

Kasutaja saab läbi veebibrauseri ning vaheserveri suhelda kasutajaliidesega. Kasutajaliides suunab pärast edukat sisselogimist kasutaja kalkulaatorilehele, kus on võimalik läbi viia arvutusi ning näha nende vormistatud tulemusi. Selleks suhtleb kasutajaliides otse kahe teise teenusega: autentimisteenus ning kalkulaator.

Autentimisteenus hoiustab kasutajate infot ning võimaldab teistel teenustel läbi API otspunktide tegeleda kasutajahaldusega: registreerimine, sisse- ja väljalogimine, parooli

muutmine ja kasutajate info pärimine. Läbi autentimisteenuse on võimalik pärida kasutaja rolle ning õigusi, et nende põhjal tegevustele vajalikke piiranguid seada. Näiteks ei tohi lubada ühe kliendi kasutajale ligipääsu teise kliendi seadistustele ning vastupidi.

Kalkulaator vastutab süsteemis kahe olulise osa eest: kalkulatsioonide läbiviimine ning kasutajate seadistuste (sisendid ning väljundid) haldamine, sealjuures nende hoiustamine andmebaasis. Küll aga ei pea erinevalt varasemast tegelema kalkulaator PV-ennustuste loomise ega optimaalse strateegia leidmisega. Selle jaoks on kasutada eraldiseisvad teenused. Selline ülesehitus muudab rakendust modulaarsemaks ning tagab parema koodi hallatavuse. Lisaks loovad eraldiseisvad teenused võimaluse tulevikus PV-ennustuste ning optimeerija tükke kasutada ka teistes rakendustes või avada need otseühenduseks klientidele liidestamiseks.

Lisaks tuumsüsteemi- ning autentimisteenustele võetakse kasutusele ka tugiteenused seireks ning masinõppe mudelite haldamiseks (mudeliregister). Seiremehhanism, mille ülesanne on reaalajas ülejäänud teenuste koormuse jälgimine ning anomaaliate korral teavituste saatmine, suurendab märkimisväärselt süsteemi töökindlust ning aitab kaasa ka piisava jõudluse tagamisel [32]. Mudeliregister võimaldab versioneerida, seirata ning ülejäänud rakenduse loogikast eraldiseisvalt uuendada pidevalt täienevaid masinõppemudeleid, mis lihtsustab veelgi süsteemihaldust [33].

Mikroteenustevahelises suhtluses kasutatakse üldjuhul ühte kahest mustrist: koreograafiat või orkestreerimist [34]. Neist esimene kasutab kommunikeerimiseks sõnumimaaklerit ehk vahekihti, kuhu teenused saavad sõnumeid (näiteks töökäske või -tulemusi) edastada ning neid sealt ka lugeda. Sellise ülesehitusega ei pea teenused üksteisest mitte midagi teadma, mis vähendab küll teenustevahelist sidusust, kuid loob keerulisema arhitektuuri. Orkestreerimisel on kasutusel nii-öelda kontroller-teenus, mis vahendab suhtlust (näiteks API kaudu) ülejäänud teenuste vahel. See tähendab, et teenused on küll üksteisest sõltuvad, kuid tagab lihtsama ülesehituse ning jälgitavuse. Tulenevalt orkestreerimise lihtsusest kasutatakse just seda, küll aga ei looda eraldi kontroller-teenust – seda rolli täidab keskse teenusena kalkulaator.

Teenuste pakendamiseks kasutatakse konteinerdust. See võimaldab rakendust juurutada olenemata kasutatava arvutusressursi omadustest, olgu selleks eraldatud riistvara või virtuaalmasin pilves. Üks mikroteenus võib koosneda mitmest konteinerist, näiteks luuakse kalkulaatorile üks konteiner API ning teine andmebaasi jaoks.

2.2. Kasutatavad tehnoloogiad

Kasutatavate tehnoloogiate alapeatükis antakse ülevaade lahenduse loomiseks kasutatavatest spetsiifilistest tehnoloogiatest. Muuhulgas tuuakse välja tehnoloogiate taust, selgitatakse nende tööpõhimõtteid, kirjeldatakse nende eeliseid ning põhjendatakse, miks on otsustatud just nende kasuks. Tehnoloogiad on jaotatud neljaks: tausta-, tuumsüsteemi- ja tugiteenuste ning kasutajaliidese tehnoloogiad.

2.2.1. Taustateenused

Docker on 2013. aastal loodud konteinerdustööriist [35]. Docker kasutab konteinerite loomiseks eelgenereeritud malle ehk pilte (ingl *images*), mida on kerge jagada (näiteks läbi pildihoidla) ning laiendada [36]. Pilt sisaldab endas kõike vajalikku konteineri jooksutamiseks, sealhulgas rakendust ning vajalikke teeke. Dockeri deemonit (ingl *daemon*) kasutades saab pildist luua käitava konteineri, mis toimib sõltumata aluskeskkonnast identselt. Ühest pildist võib käitada kuitahes mitu isoleeritud konteinerit ning ühes masinas saab samaaegselt joosta mitmeid konteinereid. See teeb nii arendamise kui juurutamise eri platvormidel lihtsamaks ning robustsemaks. Dockeri ametlikus pildihoidlas Docker Hub on saadaval üle 14 miljoni eelseadistatud pildi eri tarkvarade käitamiseks [37]. Docker on kasutatuim konteinerdustehnoloogia maailmas [38]. Vajadusest lihtsa vaevaga rakendust käitada nii erinevates arendajate masinates kui ka mitmetes toodangukeskkondades, kasutatakse teenuste pakendamiseks Dockerit.

Docker Compose on tööriist, mis võimaldab hallata mitmetest konteineritest koosnevaid rakendusi [39]. See annab arendajale võimaluse teenuste, nende võrgu, andmehoidlate ning lisaparameetrite konfigureerimiseks ühest kesksest YAML failist. Konfiguratsioonifaili põhjal suhtleb Compose Dockeri deemoniga, võimaldades ühe käsuga muuseas kõikide teenuste käivitamise, peatamise ning seire. Lisaks lihtsustatud haldusele optimeerib Docker Compose läbi konteinerite taaskasutuse ning puhvrisüsteemi ka rakenduse arenduskiirust. Tulenevalt arendatava rakenduse teenuste arvust kasutatakse töövoo lihtsustamiseks Docker Compose'i.

Nginx on HTTP veebiserver, mida saab kasutada ka pöördproksina, koormusjaoturina ning staatilise sisu puhverdamiseks [40]. Teisisõnu on Nginx server kliendi ning rakenduse vahel, vahendades ning kontrollides nende kahe vahelist liiklust. Netcrafti 2025. aasta märtsi uuringu põhjal on see kasutusel viiendikus kõikidest veebilehtedest, tehes selle Apache ees kõige populaarsemaks veebiserveriks [41]. Lisaks eelnevalt mainitud võimalustele saab Nginxi kasutada ka kliendi poolt saadetud päringute piiramiseks (ingl *rate limiting*). Need

võimalused tagavad rakenduse parema turvalisuse ning töökindluse [40]. Arendatava rakenduse veebipõhisest loomust tulenevalt on vajadus veebiserveri järele. Tänu selle turvalisus- ning töökindlusvõimetele kasutatakse selle jaoks Nginxi.

2.2.2. Tuumsüsteemi teenused

Python on 1991. aastal Guido van Rossumi poolt arendatud üldotstarbeline interpreteeritud programmeerimiskeel. Python on disainitud olema võimalikult kergesti õpitav ning programmid selles kiiresti arendatavad, mis teeb keele lihtsasti loetavaks ning intuitiivseks [42]. Alates 2023. aastast on Python TIOBE'i indeksi põhjal kõige populaarsem programmeerimiskeel maailmas [43]. Python on avatud lähtekoodiga ning aktiivse kogukonnaga [42]. Lisaks laiaulatuslikule standardteegile on saadaval ka ametlik kolmandate osapoolte tarkvarahoidla Python Package Index [44]. Just kolmandate osapoolte tööriistad nagu näiteks Jupyter ja Pandas on teinud Pythonist laialdaselt levinud programmeerimiskeele andmeteaduses [45]. Tulenevalt projekti tihedast seotusest andmeteadusega ning STACCi töötajate kompetentsist, on tagaliidese arenduseks kasutatud Pythoni programmeerimiskeelt.

FastAPI on Pythoni raamistik rakendusliideste loomiseks, mis on ehitatud olema lihtne, intuitiivne ning robustne [46]. Sarnaselt alternatiivile Flask ning erinedes Djangost on see oma olemuselt mikroraamistik: puuduvad lisafunktsionaalsused nagu näiteks andmebaasiintegratsioon. Neid on võimalik lisada vastavalt vajadusele teiste teekide kaudu [47]. Stack Overflow iga-aastase uuringu põhjal on FastAPI üks tahetuim veebitehnoloogia tarkvaraarendajate seas [48], TechEmpoweri kiirustestide põhjal on see kiireim Pythoni veebiraamistik [49]. Tulenevalt oma lihtsusest ning kiirusest kasutatakse EEDA teenuste rakendusliideste loomiseks FastAPIt.

2.2.3. Kasutajaliides

Algselt Facebooki poolt loodud JavaScript programmeerimiskeele kasutajaliideste teek React on 2024. aasta seisuga saanud üheks kasutatuimaks veebitehnoloogiaks maailmas [50, 51]. See põhineb komponentidel ehk selgesti eristatavatel kasutajaliidese tükkidel (näiteks lehekülje päis, jalus või spetsiifiline nupp), mida saab kombineerida ning taaskasutada veebilehe loomiseks [52]. Komponendid defineeritakse HTML-i sarnase märgistuskeelega JSX. React on just nimelt teek, mitte veebiraamistik. See võimaldab komponentide defineerimist, kuid ei sisalda marsruutimist, andmete laadimist ega muud veebiraamistikule olulist. Selle populaarsuse, modulaarsuse ning lihtsa märgistuskeele toe tõttu kasutatakse EEDA kasutajaliidese loomiseks Reacti.

Next.js on Reacti teegil põhinev raamistik, mis võimaldab täiemahuliste veebirakenduste arendamist. See lisab kasutajaliidese komponentidele marsruutimise, esituskihi, andmete laadimise toe ning optimeeritud staatilise sisu nagu piltide ja kirjatüüpide laadimise [53]. Lisaks võimaldab Next.js vahetarkvara (ingl *middleware*) kasutamist [54], mis lihtsustab olulisel määral kasutajate autentimist ning autoriseerimist [55]. Tulenevalt vajadusest lisada EEDA-le autentimisvõimekus, kasutatakse veebiraamistikuna just Next.js-i.

2.2.4. Tugiteenused

Teenuste töö seiramine suurendab olulisel määral süsteemi töökindlust [32]. EEDA süsteemis on vajadus seirata Dockeri konteinerite tööd, mida saab teha Google'i arendatud tööristaga cAdvisor ehk Container Advisor [56]. See kogub reaalajas andmeid käimasolevate konteinerite tööst, sealhulgas nende protsessori- ning mälukasutusest. cAdvisorit ennast saab jooksutada Dockeri konteinerina ning vajab seeläbi minimaalselt seadistamist (vt joonis 7). cAdvisor pakub kasutajaliidest reaalaja ning lähimineviku andmetest ülevaate saamiseks. Lisaks liidestub see kergesti mitmete väliste andmebaasidega, võimaldades seeläbi pikemaajalisemat andmete salvestust ning integratsiooni muude kasutajaliidestega [57].

```yaml
monitoring-cadvisor:
  container_name: eeda_monitoring_cadvisor
  image: gcr.io/cadvisor/cadvisor:v0.49.1
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
  restart: always

monitoring-prometheus:
  container_name: eeda_monitoring_prometheus
  build:
    context: ../eeda_monitoring/prometheus
  restart: always
  depends_on:
    monitoring-cadvisor:
      condition: service_started
```

Joonis 7. cAdvisori ja Prometheusi käivitamiseks vajalik Docker Compose'i konfiguratsioon.

Üheks cAdvisoriga liidestatavaks tarkvaraks on Prometheus [57]. Prometheus sisaldab endas nii aegridadel põhinevat andmebaasisüsteemi kui ka andmete pärimiseks kasutatavat päringukeelt PromQL. See on mõeldud just erinevate mõõtude salvestamiseks ning nende jälgimiseks üle aja, mis teeb selle antud olukorras sobivaks valikuks. Prometheus kasutab

andmete kogumiseks koorimist (ingl *scraping*), pärides kindla ajaintervalli tagant andmeid seadistatud allikatest [58]. cAdvisori liidestamise lihtsust Prometheusiga illustreerib joonis 8.

```
scrape_configs:
  - job_name: cadvisor
    scrape_interval: 5s
    static_configs:
      - targets:
          - monitoring-cadvisor:8080
```

Joonis 8. cAdvisori kui Prometheusi andmeallika seadistamiseks vajalik konfiguratsioon.

Salvestatavate andmete kuvamiseks ning väärtuslikuks kasutamiseks on tarvis kasutusele võtta koondpaneel. Tulenevalt sisseehitatud Prometheusi liidestusest ning paljudest seire lisavõimalustest nagu näiteks hoiatusteadete saatmine, sobib selleks Grafana. Grafana võimaldab erinevate andmeallikate põhjal interaktiivsete näidikulaudade (ingl *dahsboard*) loomist. Tänu aktiivsele kogukonnale on vabavaraliselt saadaval mitmed eelseadistatud Grafana näidikulauad [59]. Sealhulgas on ka Prometheuse toega cAdvisori andmete näidikulaud[2], mida autor töös kasutab. Lisaks võimaldab Grafana andmete põhjal automaatsete teadete saatmist[3], mis tagab veelgi parema süsteemi jälgitavuse.

MLflow on laiaotstarbeline masinõppetööriist. Selle eesmärk on muuta masinõppemudelite arendamise elutsükkel hallatavaks, jälgitavaks ning reprodutseeritavaks. See võimaldab mudelieksperimentide automaatset salvestust (parameetrid, andmestikud ning täpsusmõõdud), mudelite standardiseeritud pakendamist kui ka spetsiifiliste mudelite versioneerimist läbi mudeliregistri. MLflow mudeliregister võimaldab nii rakendusliidese kui ka Pythoni teegi kaudu välistel süsteemidel keskkonnast sõltumatult treenitud mudeleid ennustuste loomiseks kasutada [60]. Tulenevalt STACCi varasematest kogemustest MLflow kui mudeliregistri kasutamisega, võetakse see kasutusele ka EEDA süsteemis.

Ühtlustamaks süsteemiüleselt kasutatavaid tehnoloogiad, kasutatakse autentimisteenuse loomiseks samuti FastAPIt. Tulenevalt autentimisega seotud keerukustest ning selle olulisusest turvalisuse tagamisel ei arendata kasutajaloogikat välja käsitsi, vaid kasutatakse selleks juba olemasolevat teeki FastAPI Users [61]. Teek võimaldab kohandatavate kasutajamudelite loomist ning pakub kasutamiseks mitmeid autentimise ning

[2] Eelseadistatud Grafana näidikulaud cAdvisori andmete visualiseerimiseks. https://grafana.com/grafana/dashboards/19908-docker-container-monitoring-with-prometheus-and-cadvisor/.
[3] Grafana teadetesüsteem. https://grafana.com/docs/grafana/latest/alerting/.

kasutajahaldusega seotud rakendusliidese otspunkte. Seadistatavad on ka teegi poolt kasutatav andmebaas, autentimis- ning andmete edastusmeetod.

2.3. Teenuste ülesehitus

Teenuste peatükk kirjeldab täpsemat lähenemist tuumsüsteemi jaoks vajalike teenuste ülesehitusest. Tuuakse välja nende nõuded ning võimalikud probleemsed ning süvenemist vajavad kohad. See loob aluse edasisele süsteemi arendusele.

2.3.1. PV-ennustused

EEDA on mõeldud päikeseenergia investeeringute tasuvusarvutusteks. Kodust päikeseparki planeerides saab arvutustes kasutada küll näiteks ajaloolist elektritarbimise mustrit, kuid päikesepaneelide tootlikkust tuleb ennustada. Selleks on üheks süsteemi komponendiks PV-ennustuste teenus. Praegune ennustusmootor saab sisenditeks planeeritava pargi asukoha koordinaadid ning ennustust vajavad kuupäevad. Nende põhjal laetakse juurde ilmaennustused ning kasutatakse masinõppemudeleid ennustuste genereerimiseks. Masinõppemudelid käivad kaasas rakenduse koodibaasiga ning ennustusmootor on integreeritud otse EEDA koodi.

Uue süsteemi tarbeks luuakse ülejäänud rakendusest eraldiseisev API-l põhinev PV-ennustuste teenus. Sarnaselt olemasolevale loogikale võtab API otspunkt sisenditena vastu päikesepargi koordinaadid ning ennustatava ajavahemiku ning pärib nende põhjal välisest teenusest ilmaennustused. Ilmaennustusi kasutatakse masinõppemudelite sisenditena, kuid erineb mudelite kasutusloogika. Nimelt võetakse selle jaoks kasutusse mudeliregister, läbi mille API ennustusi küsitakse.

Mudeliregistrit ennast võib vaadata kui API-liidestusega versioonihaldusteenust masinõppemudelite jaoks. Läbi selle on võimalik ennustuste tegemiseks pärida viimast versiooni vajaminevast mudelist. Lisaks tegeleb mudeliregister ka mudelite pakendamisega, ehk mudelit kasutav süsteem ei pea omama ühtegi mudeli jooksutamiseks vajalikku sõltuvust. See võimaldab luua masinõppemudeleid andmeteadlasele sobilikus keskkonnas, neid omavahel kergesti võrrelda ning juurutada eraldiseisvalt teenustest, mis mudeleid kasutavad.

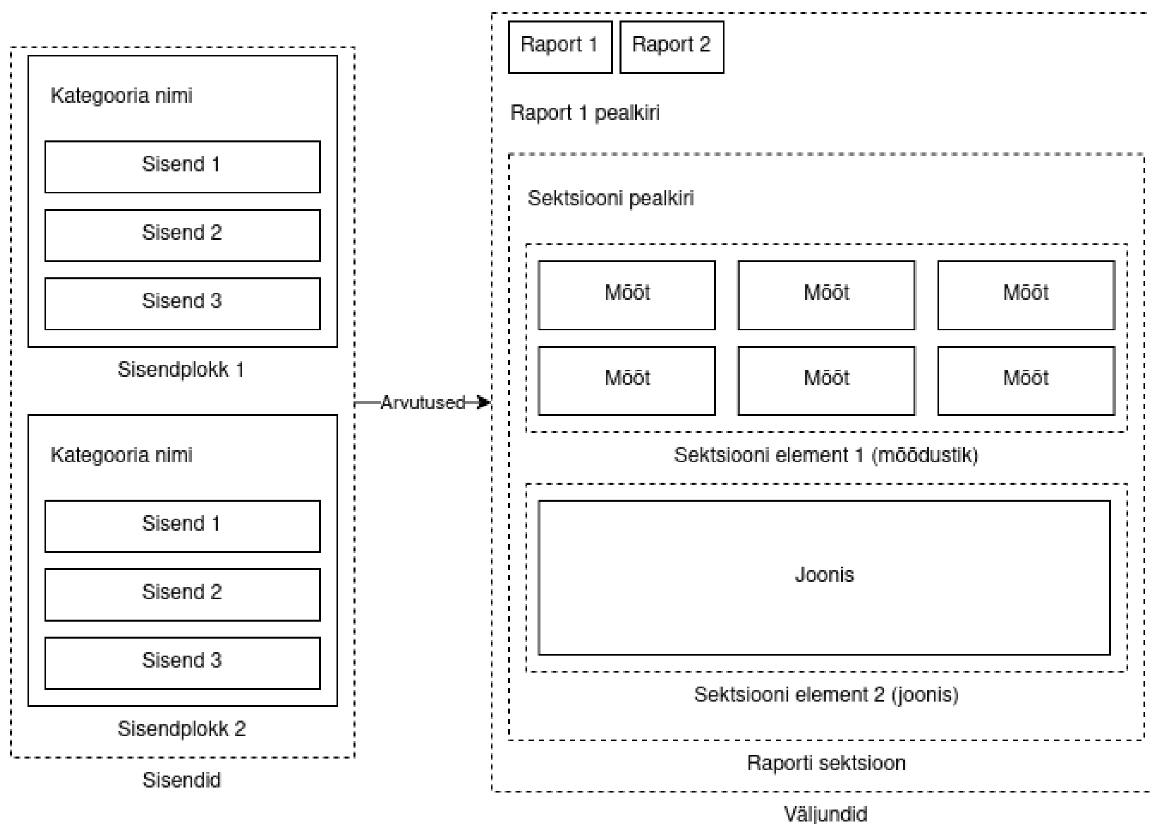
PV-ennustuste kui API-põhise teenuse loomine võimaldab tulevikus selle liidestamist uute rakendustega. Lisaks jääb avatuks äriline võimalus müüa PV-ennustuste API-t kui omaette teenust, mida kliendid oma süsteemidesse liita saavad.

2.3.2. Kalkulaator

Kalkulaator kui teenus haldab nii kasutajate kalkulaatori seadistusi kui ka sisendite põhjal kalkulatsioonide (tasuvusarvutused, strateegia optimeerimine) läbiviimist. Kalkulaatori seadistused saab jaotada kaheks: sisendid ning väljundid. Mõlema seadistuse elemendid sisaldavad endas suurel hulgal äriloogikat, mistõttu on vajalik nende defineerimine kooditasandil, selmet kasutada näiteks staatilisi andmebaasikirjeid. Mõlemate elementide muutmine ning lisamine peab olema ka võimalikult lihtne, sest seda tuleb teha tihti.

Varasemast on sisendid jaotatud loogilistesse kogumitesse ehk sisendplokkidesse (ingl *input block* või lihtsalt *block*). Plokid on loodud reaalsete energiasüsteemi komponentide, näiteks päikesepaneelid, energiasalvesti ning elektriauto põhjal. Ühes plokis olevad sisendid on seadistatud manuaalselt, et need oleksid omavahel sobivad ning sisult mitte-kattuvad ja need võivad klienditi palju erineda (vt lisa 1). Sisendid ise jaotuvad eri tüüpide, nagu näiteks arvu- või tekstilahter, liugur, märkeruut (ingl *checkbox*), raadionupp ning rippmenüü vahel, mida Streamlit vaikimisi toetab. Lisaks on enamikel sisenditel kasutusel väärtuste valideerimine, näiteks arvulise sisendi jaoks on määratud lubatud väärtuste vahemik.

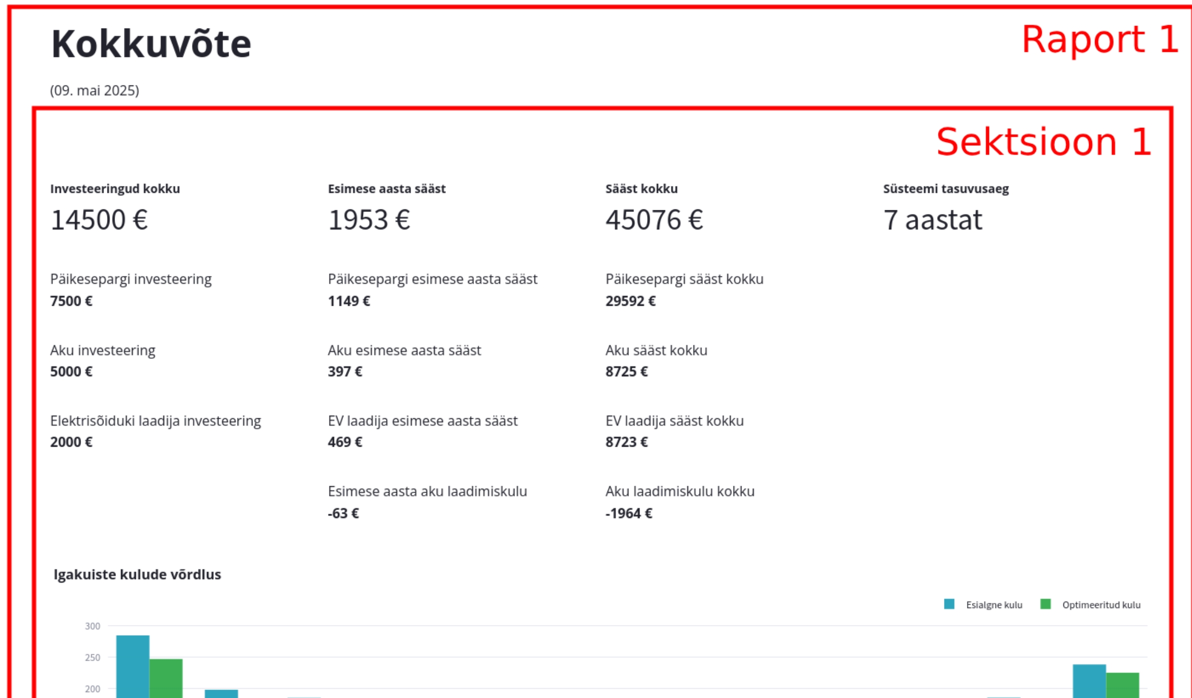
Tulenevalt kirjeldatud sisendite struktuuri sobivusest uue süsteemi kontekstis, ei ole vajalik olemasolevat plokkidepõhist lahendust palju muuta (vt joonis 9). Küll aga puudub peatükis 1.1 kirjeldatud sisendplokkide kategooria kontseptsioon, sest klientide jaoks on koodibaasi muudetud selliselt, et iga kategooria kohta on seadistatud täpselt üks sisendplokk ning neid saab seetõttu lugeda sünonüümideks. Uus süsteem peab aga võimaldama ühes kategoorias mitme erineva sisendploki defineerimist. Sealjuures tuleb arvestada, et kliendile saab ühest kategooriast seadistada maksimaalselt ühe sisendploki. Näiteks võib tarbimiskategoorias olla seadistatud üks plokk, mis laseb kasutajal üles laadida eelmise aasta tunnipõhise tarbimisfaili, ning teine plokk, mis võimaldab aastase kogutarbimise sisestamist. Ühel kliendil võib olla korraga seadistatud vaid üks kahest variandist. Seega tuleb lisada sisendplokkide kohale kategooria loogika, mis hoiaks endas plokkideülest informatsiooni (nagu näiteks kategooria nimi) ning kõikvõimalikke selle kategooria jaoks defineeritud sisendplokke.



Joonis 9. Kalukulaatori seadistuste struktuur.

Lisaks ei ole enam võimalik Streamliti sisseehitatud sisendobjektide nagu näiteks teksti- või numbrilahtrite kasutamine, mis võimaldasid varasemalt vähese vaevaga otse Pythoni koodist kasutajaliidest luua ning sisendeid ka valideerida. Selle asemele tuleb luua uus sisendite defineerimise loogika, mis võimaldaks ühelt poolt nii sisendite jadastust (ingl *serialization*), et need sobivas struktuuris kasutajaliidesele edastada, teisalt aga ka sisestatud väärtuste vastu võtmist ning nende valideerimist eelnevalt seadistatud reeglite vastu.

Varasemalt on kalkulaatori väljundite jaoks loodud raportitepõhine struktuur (vt joonis 10). Iga kalkulatsioon saab genereerida ühe või mitu raportit, mis koosneb omakorda sektsioonidest. Üks sektsioon moodustab loogilise grupi väljunditest, näiteks annab ülevaate elektrisõiduki laadimisest või päikeseenergia tootlikkusest. Sektsioonidel võib olla oma pealkiri ning need sisaldavad eri tüüpi elemente: mõõdud, joonised, tabelid. Tasub märkida, et elemendid (mõõdud, joonised, tabelid) on ainukesed arvutustest sõltuvad osad raportis, sektsioonid ning ka raport ise on vaid abstraktsed struktuuri loomiseks mõeldud vahendid.



Joonis 10. Kolme raportiga EEDA väljund.

Kliendipõhiseid koodihoidlaid kasutades on raporti elementide arvutamine ning struktuuri loomine käinud käsikäes. On täpselt teada, milliseid mõõte on vaja arvutada ning kuhu need paigutada, seega saab seda teha lineaarselt. Uus süsteem peab olema aga oluliselt paindlikum: igale kliendile peab saama raporteid dünaamiliselt seadistada, sealhulgas määrata sektsioonide järjekorda, elementide järjekorda sektsioonidesiseselt ning ka mõõtude paiknemist mõõdustikes.

Elemente seadistades tekib aga probleem: sisendplokkidel võivad puududa vajalikud sisendid teatud raporti elementide välja arvutamiseks. Näiteks kui sisendplokis puudub informatsioon päikesepaneelide investeeringu suuruse kohta, ei saa välja arvutada investeeringu tasuvusaega. Seega tuleb luua raporti elementide juurde nii-öelda sõltuvuste loogika. See loob võimaluse kontrollimaks, kas kindlate seadistatud sisendite puhul on antud element arvutatav või mitte. Kontrollmehhanismi tuleb rakendada nii kliendi seadistuse loomisel kui ka jooksvalt elemente arvutades, et tagada rakenduse töökindlus.

Lisaks on olemasoleval lahendusel väga tähtis elementide arvutamise järjekord, sest mõne mõõdu arvutatud väärtust kasutatakse omakorda teise mõõdu arvutamiseks. Kuna sellised sõltuvused aga selgelt defineeritud pole, tekitab see arusaamatusi. Lisaks arvutatakse lihtsuse

huvides sama sisuga väärtusi mitmes eri kohas, mis loob vägagi veaohtliku situatsiooni. Selle lahendamiseks tuleb loodavale sõltuvuste loogikale lisada ka mõõtude kui sõltuvuste tugi. Kusjuures tuleb arvestada, et kuvataval elemendil võivad olla sellised mõõtudest sõltuvused, mis on küll arvutatavad, aga mida ise raportis ei kuvata. Lisaks peab tagama, et mõõte arvutatakse õiges järjekorras ning ei tekiks kahe mõõdu omavahelist sõltuvust (ingl *circular dependency*).

2.3.3. Optimeerija

Optimeerija tööülesandeks on etteantud parameetrite puhul tunnipõhise optimaalse strateegia leidmine ning selle koos vajalike lisaandmete tagastamine. Strateegia leitakse rekursiivselt erinevaid tegevusi (ingl *action*) läbi proovides. Teisisõnu leitakse selline tegevuste järjestikune teekond, mille puhul rahaline võit on maksimaalne (või kahju minimaalne). Tegevusi on varasemast defineeritud kuusteist tükki. Kuivõrd optimeerimisalgoritm, sealhulgas tegevused, on juba varasemalt defineeritud ning neid on võimalik samal kujul ka edasi kasutada, ei keskenduta antud töös nende spetsiifikale.

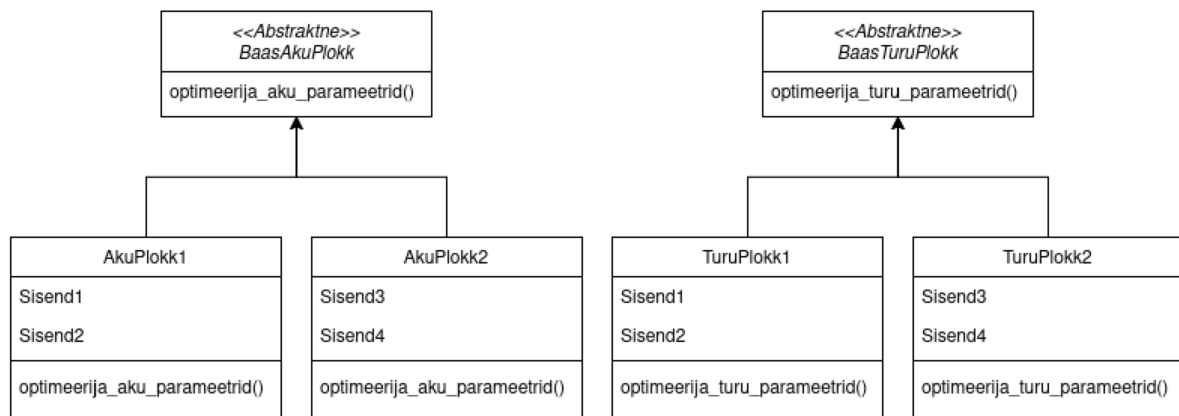
Küll aga on Streamlitipõhises rakenduses optimeerija loogika põimitud tugevalt kogu ülejäänud rakenduse koodiga. Näiteks antakse optimeerijale kasutamiseks ette kõik kasutaja sisendid ning nendest tuletatud andmed, kuid need erinevad klienditi. Seetõttu on ka optimeerija koodi kliendipõhiselt kohandatud, mida tuleb optimeerija kui tsentraalse teenuse kontekstis muuta. Lisaks on olemasoleva lahenduse puhul keeruline jälgida milliseid sisendeid ning andmeid optimeerija üldse vajab, mida tuleb samuti uurida.

Optimeerija tööloogika põhjalikuma analüüsi tulemusena selgub, et parameetrid on võimalik iga olemasoleva kliendi seadistuse jaoks viia ühtlustatud kujule. Täpsemalt saab optimeerija parameetrid jaotada kolmeks:

1. tunnipõhised andmed: iga optimeerimisperioodis oleva tunni elektri ostmis- ning müügihind, tarbimine ja PV-tootlikkus;
2. aku parameetrid: mahutavus, laadimiskiirus, päevane laadimislimiit ning mahutavuspõhine hind (ingl *levelized cost of storage*);
3. turu-/võrgu parameetrid: ostmise ja müümise võrgutasud ning limiidid.

Kliendipõhiste sisendite transformeerimiseks optimeerija jaoks vajalikeks parameetriteks saab kalkulaatoris kasutada sisendplokkide abstrahheerimist (vt joonis 11). Nimelt on võimalik luua iga sisendkategooria plokkide jaoks abstraktne ülemklass, mis kirjeldab optimeerija parameetrite eraldamiseks vajaminevat abstraktset meetodit. Meetod omakorda

määrab ära tagastatavate parameetrite tüübi. See tähendab, et näiteks uue aku kategooria sisendploki loomisel tuleb defineerida ka meetod, mis tagastaks eelnevas loendis toodud aku parameetrite väärtused. Selline lahendus ei eelda sisendplokkidelt kindlate sisendite olemasolu, vaid lubab väärtuste eraldamiseks kasutada vajalikke transformatsioone.



Joonis 11. Sisendplokkide klassipõhine ülesehitus koodis.

Vajaminevate optimeerija parameetrite saamiseks tuleb luua abstraktsed meetodid aku ning turu plokkidele. Lisaks tuleb samasugust abstrahheerimist kasutada ka tunnipõhiste andmete eraldamiseks. Täpsemalt peab olema iga tarbimisplokk võimeline tagastama tunnipõhised tarbimisandmed, iga päikesepargiplokk tootlikkusandmeid ning iga turuplokk elektrihindu.

3. Tulemused

Tulemuste peatükk võtab kokku metoodika peatükis kirjeldatud süsteemi ning teenuste arenduse. Käsitletakse nii tehniliselt keerukamaid kohti kui ka ilmnenud puudusi ning nende lahendusi. Et võrrelda valminud lahendust varasema süsteemiga, sisaldab peatükk ka nende kahe võrdlust. Sealhulgas tuuakse välja nii sisulised aspektid kui ka rakenduste majutuskulud ning jõudlustestid.

Süsteemi lähtekood on ärilistel kaalutlustel privaatne. Ligipääsu saamiseks võtta ühendust e-kirja teel aadressil karl.kevin.ruul@ut.ee.

3.1. Arendatud mikroteenused

Kõige keerulisemaks kohaks süsteemi üleviimisel osutus kalkulaatori mikroteenuse ehitamine. Sealne väljakutse seisnes konfigureeritavate sisendite ning väljundite võimekuse loomises. Sealjuures tuli ka lahendada küsimus, kuidas Pythoni koodis defineeritud äriloogikat sisaldavatele klassidele oleks võimalik töökindlalt andmebaasist viidata. Lisaks tuli süsteemi ning koodi kasvades tegeleda selle korrapärase struktureerimisega, mis samuti antud alapeatükis välja tuuakse.

3.1.1. Sisendite defineerimine

Metoodika peatükis on paika seatud sisendite hierarhia:

1. on fikseeritud hulk sisendplokkide kategooriaid, näiteks tarbimine, aku, päikesepark, elektriauto kategooriad;
2. igas kategoorias on üks või mitu sisendplokki, näiteks tarbimise kategoorias üks plokk, mis lubab üles laadida ajaloolise tarbimisfaili ning teine, mis laseb määrata aastase kogutarbimise ning hoone tüübi;
3. igas sisendplokis on omakorda defineeritud sisendid, mida kasutajale kuvatakse ning kust vajalikku infot ploki tasemel eraldatakse.

Tulenevalt vajadusest neid pidevalt lisada ning muuta, peab kõigi kolme defineerimine olema võimalikult lihtne. Näiteks tähendab see, et uue kategooria, ploki või sisendi loomisel ei oleks tarvis andmebaasi muuta ega sinna vastavat kirjet lisada. Selle saavutamiseks hallatakse kogu nende loogikat koodis. Lisaks peaks võimalikult suur osa loogikast olema taustal defineeritud nii, et uut spetsiifilist objekti luues tuleks võimalikult vähe koodi lisada. Selle saavutamiseks võeti palju eeskuju Django raamistiku veebivormidest [62]. Täpsemalt koosnes sisendite loogika defineerimine mitmest sammust.

Esmalt loodi baasklass *BaseInput* kõikide defineeritavate sisendite jaoks. Sarnaselt Django *Field* klassile [63] tegeleb *BaseInput* sisendite metainfo hoidmisega: mis on sisendi kuvatav tekst (ingl *label*), kas see on kohustuslik jne. Lisaks defineerib *BaseInput* meetodid jadastuseks, väärtuse valideerimiseks ning -puhastamiseks. Küll aga ei hoia *BaseInput* informatsiooni kasutaja poolt sisestatud väärtuse kohta, tegeledes vaid selle valideerimisega. Baasklassist loodi tütarklassid nagu näiteks *TextInput*, *FloatInput* ja *SliderInput*, mis lihtsustavad vastavat tüüpi sisendite loomist lisades vajalikke valideerimistingimusi ning metainfo välju.

Teiseks loodi baasklass *BaseInputBlock* erinevate sisendplokkide loomiseks. Plokis defineeritakse vajalikud sisendid luues *BaseInput* tütarklasside isendid. Selle jaoks võeti eeskuju Django raamistiku vormidest [62], võimaldades plokis olevaid sisendeid defineerida *BaseInputBlock* klassiatribuutidena (vt joonis 12). Atribuudid kogutakse kokku kasutades metaprogrammeerimist [64]. Täpsemalt on määratud *BaseInputBlock*ile metaklass *DeclarativeInputsMetaclass*. See leiab kõik klassi atribuutidena defineeritud *BaseInput*-tüüpi väärtused (sealhulgas ka ülemklassidest) ning salvestab need enne ploki loomist ühisesse paisketabelisse *declared_inputs*. Paisktabelit kasutades on hilisemalt lihtne nii spetsiifilist sisendit leida kui ka üle kõikide sisendite käia. See võimaldab sisendplokki jadastada ning etteantud väärtusi valideerida. Sellest tulenevalt hoitakse kasutaja sisestatud väärtusi nende olemasolul just *BaseInputBlockis*.

```
@BatteryBlockCategory.register()  Karl Kevin Ruul *
class HomepageBatteryBlock(BaseBatteryBlock):

    investment = IntegerInput(
        label='Battery investment', unit='EUR', default_value=5_000, min_value=0
    )
    capacity = FloatInput(
        label='Battery capacity', unit='kWh', default_value=9.6, min_value=0
    )
    c_rate = SliderInput(
        label='Battery C-rate', default_value=0.5, min_value=0.1, max_value=1.0, step=0.05
    )
    battery_cycles = IntegerInput(
        label='Battery cycles', default_value=7_000, min_value=0
    )
    charge_range = SliderInput(
        label='Battery charge range', default_value=(5, 95), min_value=0, max_value=100, step=5
    )
    inverter_power = FloatInput(
        label='Inverter power', unit='kW', default_value=8, min_value=0
    )
```

Joonis 12. Aku sisendploki defineerimine erinevaid sisendeid kasutades.

Kolmandaks loodi baasklass *BaseInputBlockCategory* kategooriate defineerimiseks. Oma olemuselt hoiab kategooria klass vaid metainfot: mis on kategooria nimi ning kas selle väärtuste täitmine on kasutajale kohustuslik või valikuline. Läbi kategooriaklassi on saadaval ka loend kõikidest antud kategooria sisendplokkidest. Selle toimimise loogikat kirjeldatakse täpsemalt peatükis 5.1.3. *BaseInputBlockCategory* võimaldab samuti jadastust, et vajalikku infot kasutajaliidesele saata.

3.1.2. Väljundite sõltuvuste haldamine

Sarnaselt sisenditele kohandatakse ning lisatakse pidevalt ka väljundeid, mistõttu peab nende defineerimine olema samuti võimalikult lihtne. Väljundite puhul tuleb arvestada, et need on sõltuvad sisenditest. Ühelt poolt tähendab see, et väljundit defineerides peab olema võimalik viidata vajalikele sisenditele. Teisalt peab olema võimalus kontrollimaks, kas etteantud sisendite konfiguratsiooni puhul on võimalik antud väljundit arvutada. Töö käigus selgus kiiresti, et väljundid saavad ka üksteisest, täpsemalt teistest mõõtudest, sõltuvad olla. Näiteks projekti koguinvesteering sõltub erinevate kategooriate investeeringutest.

Sõltuvuste haldamiseks loodi *DependencyMixin* klass. Tegu on mix-in tüüpi klassiga [65], mille lisamine mõnele muule (väljund-)klassile lisab sellele sõltuvuste haldamise loogika (vt joonis 13). Täpsemalt lisatakse neli klassiatribuuti eri tüüpi sõltuvuste määramiseks:

1. *input_dependencies*: paisktabel, mis seab sisendkategooriale vastavusse loendi nõutud sisenditest;
2. *metric_dependencies*: nimekiri mõõtudest, mille väärtust on antud väljundi arvutamiseks vaja;
3. *soft_metric_dependencies*: nimekiri mõõtudest, millest vähemalt ühte on antud väljundi arvutamiseks vaja;
4. *optimization_dependency*: tõeväärtus määramaks, kas väljundi arvutamiseks on vaja optimeerija tulemusi.

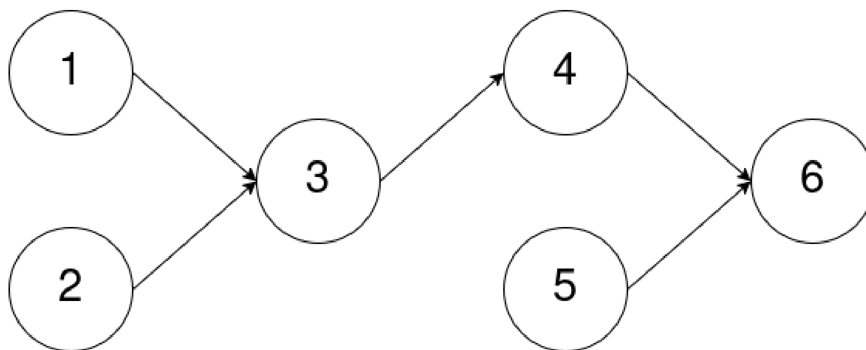
```
@MetricsRegistry.register()  Karl Kevin Ruul
class BatteryCycles(BaseIntegerMetric):
    input_dependencies = {InputBlockCategoryType.BATTERY: ['min_capacity', 'max_capacity']}
    metric_dependencies = {BatteryEnergyStored}

    @classmethod  Karl Kevin Ruul
    def get_id(cls) -> str:
        return 'battery_cycles'

    @classmethod  Karl Kevin Ruul
    def get_label(cls) -> str:
        return 'Cycles done'

    def _calculate(self) -> int:  Karl Kevin Ruul
        return self.calculation.get_metric(BatteryEnergyStored).value / (
            self.calculation.get_input_block(BaseBatteryBlock).max_capacity
            - self.calculation.get_input_block(BaseBatteryBlock).min_capacity
        )
```

Joonis 13. Sisenditest kui ka teisest mõõdust sõltuva mõõdu defineerimine.

DependencyMixin klassis on defineeritud ka funktsioon *is_calculable*, mis etteantud sisendite konfiguratsiooni ning klassiatribuutide väärtuste põhjal leiab, kas väljund on arvutatav. Funktsioon kontrollib esmalt, kas kõik nõutud sisendid on etteantud konfiguratsioonis seadistatud. Seejärel kontrollitakse, kas kõik arvutamiseks vajalikud mõõdud on arvutatavad. Väljundi arvutatavust kontrollitakse nii väljundite seadistamisel kui ka kalkulaatori töövoos enne arvutuste läbiviimist. See tagab parema veahalduse juhul, mil varasemalt seadistatud väljundite sõltuvustes on tehtud muudatusi, mis ei võimalda enam seadistatud sisendite puhul arvutusi läbi viia.

Mõõtude kui sõltuvuste lisamisega tekkis aga potentsiaalne veaohtlik koht: ei tohi juhtuda, et mõõdud üksteisest sõltuksid. Selle lahendamiseks moodustatakse mõõtudest ning nende sõltuvustest esmalt suunatud graaf. Graaf koostatakse selliselt, et igast sõltuvusena kasutatavast mõõdust väljuvad servad mõõtudesse, mis seda sõltuvusena kasutavad. Seejärel järjestatakse graaf kasutades topoloogilist sortimist [66]. Tulemuseks on selline mõõtude loend, kus iga sõltuvuseks olev mõõt tuleb järjekorras enne mõõte, millele ta sõltuvuseks on. Näide ühest sellisest graafist koos tema kõikvõimalike topoloogilise sortimise tulemustega on kujutatud joonisel 14. See aga tähendab, et topoloogiline sortimine on võimalik vaid juhul, mil suunatud graafis puuduvad tsüklid. Teisisõnu peab olema tegu atsüklilise suunatud graafiga (ingl *acyclic directed graph* ehk *DAG*). Seega kontrollitakse läbi sortimise, et

mõõdud üksteisest ei sõltuks. Lisaks annab topoloogiline sortimine sobiliku järjekorra mõõtude arvutamiseks, sest enne mõõdu arvutamist on tema sõltuvused juba arvutatud.



Joonis 14. Atsükliline suunatud graaf koos kõikide topoloogilise sortimise võimalustega.

3.1.3. Äriloogika klasside registrid

Sisendid ning väljundid sisaldavad endas suurel hulgal äriloogikat ning on seetõttu loodud varasemalt kirjeldatud Pythoni klassidena. Küll aga peab kliendipõhiseks seadistamiseks salvestama andmebaasi nende kohta käivaid viiteid. Näiteks määratakse igale kliendile andmebaasis võti-väärtus paaridena vastavusse seadistatud sisendkategooria ning -plokk.

Selle jaoks loodi esmalt abstraktne klass *Identifiable*. See määrab alamklassides defineerimist vajava meetodi *get_id* klassi identifitseerimiseks. Klassi identifikaator (ID) peab olema unikaalne ning ei tohi ajas muutuda. Seeläbi on seda võimalik kasutada andmebaasis viitena seadistatud klassile. Seejärel loodi baasklass *BaseRegistry*, läbi mille on võimalik kasutada registrite disainimustrit [67]. Täpsemalt on võimalik luua registreid erinevate *Identifiable* klasside hoiustamiseks. Läbi *BaseRegistry* klassi on igale registrile loodud võimekus ID põhjal klassi isendi kättesaamiseks. Seeläbi on võimalik andmebaasis viidetena hoiustavad seadistused laadida sisse Pythoni klassidena.

Hõlpsamaks klasside registreerimiseks registrisse loodi *BaseRegistry* klassi vastav meetod Pythoni dekoraatori[4] kujul (vt joonis 13). Registreerimine toimub käitusajal ning selle toimiseks on vajalik, et koos registriga oleksid imporditud ka kõik sinna kuuluvad klassid.

[4] Pythoni dekoraatorid. https://peps.python.org/pep-0318/.

Selle jaoks kasutatakse Pythoni pakettide traditsioonilist initsiaatorfaili *__init__.py*[5]. Registriga samas paketis olevas initsiaatorfailis imporditakse kõik failid, mis sisaldavad endas registreeritavate klasside definitsioone. Initisiaatorfaili jooksutatakse automaatselt iga kord, kui midagi vastavast paketist imporditakse. See tagab, et registrit importides on sinna juba kõik klassid registreeritud.

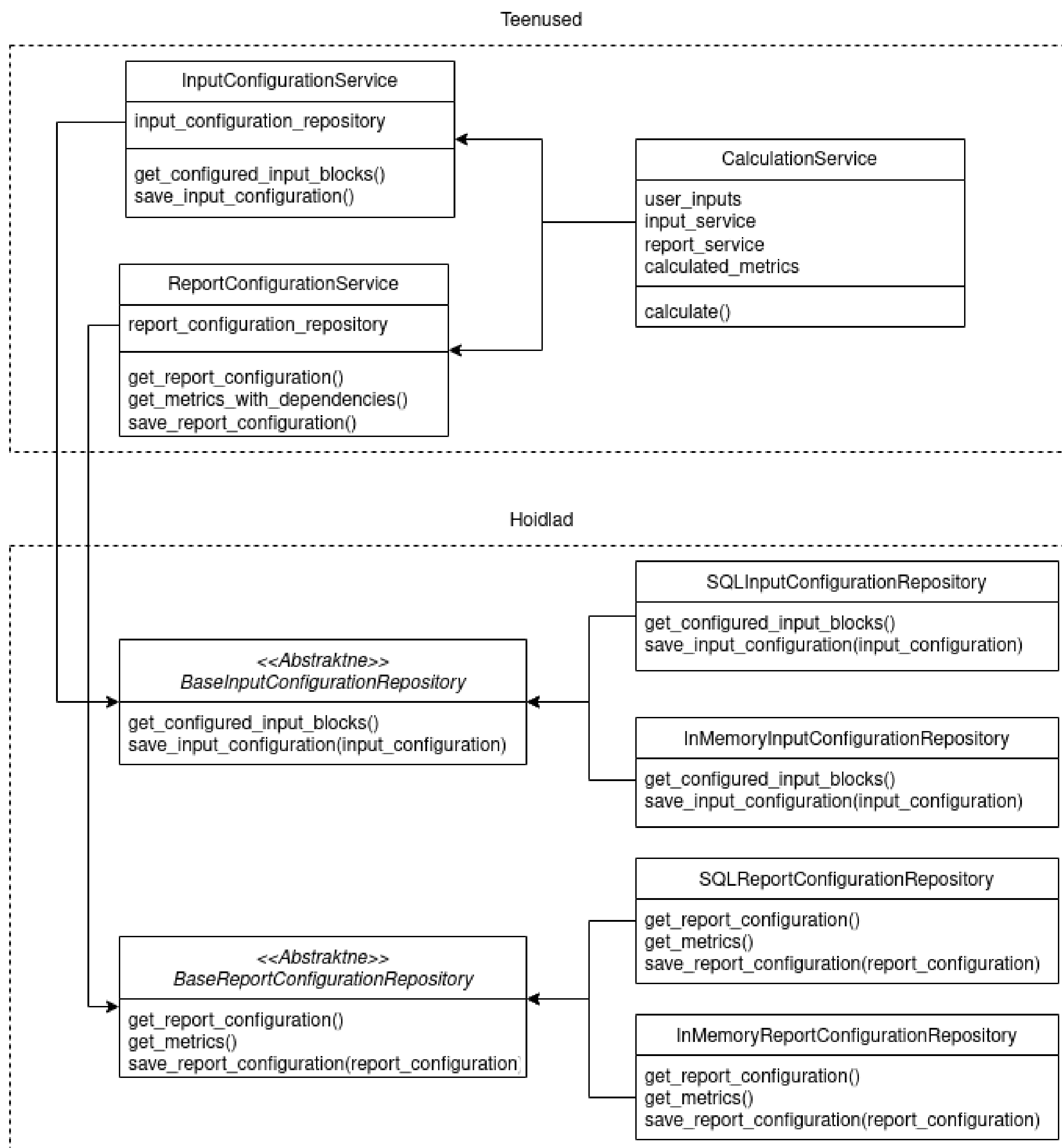
Register kui klass võimaldab juba olemasolevatest klassidest teha kerge vaevaga registrid. Näiteks on registriteks kõikide sisendkategooriate klassid (vt joonis 12), hoides endas loendit kategoorias defineeritud sisendplokkidest. Samas jääb alles ka võimalus luua täiesti eraldiseisvad registerklassid. Seda on tehtud näiteks sisendkategooriate ning kõikide väljundelementide jaoks.

3.1.4. Koodibaasi struktuur

Süsteemi mikroteenused on üles ehitatud FastAPI mikroraamistikul. Erinevalt näiteks Djangost ei sea see mingisuguseid piiranguid koodibaasi ülesehitusele. See tagab algselt küll võimalikult kiire ja vaba arendusprotsessi, kuid vajab koodibaasi suurenemisel lisa tööd selle struktureerimiseks. Kuivõrd optimeerija ning PV-ennustuste teenused on oma olemuselt lihtsamad ning vähesema koodiga, ei ole nende koodibaasides korra loomine keeruline. Küll aga kasvas kalkulaatori teenus tuhandetesse koodiridadesse ning vajas kindlamat struktuuri.

Struktuuri loomiseks võeti inspiratsiooni raamatust "*Architecture Patterns with Python*" [68]. Kasutusele võeti nii hoidlate (ingl *repository*) kui ka teenuste (ingl *service*) põhimõtted, nagu on kujutatud joonisel 15. Hoidlad vastutavad kindlat tüüpi andmete kättesaamise ning salvestamise eest. Läbi hoidlate eraldatakse üksteisest suhtlus andmebaasi ning ülejäänud koodi vahel, muutes koodi seeläbi hallatavamaks. Hoidlaid abstrahheerides on võimalik neid defineerida eri andmeallikate (näiteks andmebaas ning põhimälu) jaoks, mis lihtsustab koodi testimist. Teenuskiht haldab koodis tüüpilisi tegevusi, mis ei ole otseselt seotud äriloogikaga. Näiteks võib teenuskihis toimuda objekti salvestamine, mille korral esmalt salvestatav objekt valideeritakse, vajadusel tekkinud vigu hallatakse ning sobival korral läbi hoidla salvestatakse. See võimaldab äriloogikat sisaldavast koodist eraldada üldisema loogika.

[5] Pythoni impordisüsteem. https://docs.python.org/3/reference/import.html#regular-packages.



Joonis 15. EEDA kalkulaatori hoidlad ja teenused.

Täpsemalt loodi eraldi hoidlad ning teenused kalkulaatori sisendite ja väljundite haldamiseks. Oma ülesehituselt on need sarnased: nii sisendite kui väljundite hoidlad salvestavad ja laevad konfiguratsioone ning teenused tegelevad nende valideerimise ning vormistamisega. Hoidlaid on omakorda kaks erinevat versiooni. Üks kasutab andmete hoiustamiseks andmebaasi, teine põhimälu. Andmebaasiga ühenduvaid hoidlaid kasutatakse produktsioonisüsteemis, kus andmeid on vaja pikaajaliselt säilitada, põhimälu kasutavaid hoidlaid aga koodi testimisel, eemaldades seeläbi sõltuvuse andmebaasist. Lisaks defineeriti ka kolmas teenus arvutuste haldamiseks. See võimaldab kasutaja sisendite valideerimist ning nende põhjal vastavalt

kasutaja konfiguratsioonidele arvutuste läbiviimist. Läbi arvutusteenuse saavad väljundelemendid ligi ka sisenditele ning mõõtudele, millede väärtustest nad sõltuvad.

3.2. Vana ja uue süsteemi võrdlus

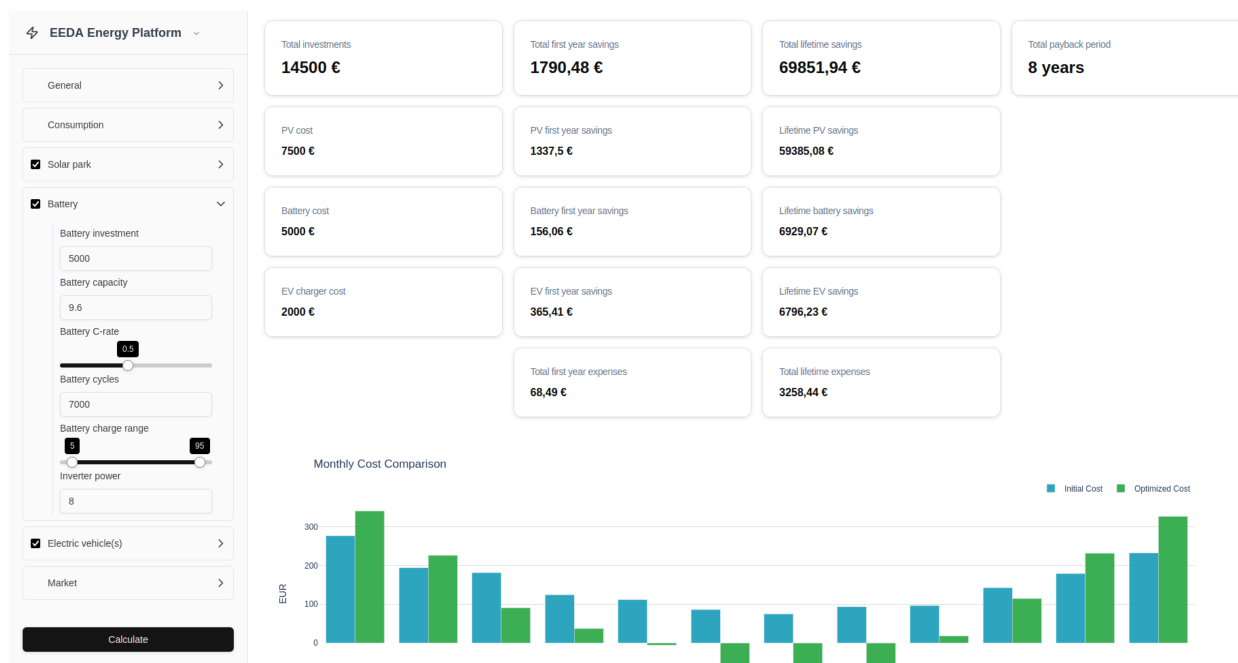
Saamaks aru, kas uus süsteem saavutas seatud eesmärgid, tuleb seda vana süsteemiga võrrelda. Võrdlus jaotatakse kolmeks osaks. Esmalt kirjeldatakse sisulisi muutusi: kuidas erineb rakenduse kasutamine ning selle arendamine. Teiseks viiakse läbi testid võrdlemaks uue süsteemi kiirust vana süsteemi kiirusega. Viimaks analüüsitakse majutuskulusid, tehes kindlaks, et uue süsteemi ülalpidamine vanast kallim ei oleks.

3.2.1. Sisulised muudatused

Suurim sisuline muudatus kasutajatele seisneb jagatud rakenduse ehk tarkvara kui teenuse kasutamises. Kui varasemalt on igale kliendile majutatud eraldiseisev rakendus aadressil klient.rakendus.ee (aadress on piltlik), siis tulevased kliendid kasutavad nende jaoks eelseadistatud rakendust aadressil rakendus.ee. Kui varasemalt oli ligipääs rakendusele üldjuhul piiratud võrguseadistustega selliselt, et ligi pääses vaid kliendi sisevõrgust, siis uus rakendus on oma loomult avalik, kuid vajab kasutamiseks autentimist.

Tulenevalt tänapäeva trendidest ei ole SaaS-i kasutamine paljudele firmadele midagi uut või vastuvõtmatut. Küll aga võib rakenduse, sealjuures ka ressrusside jagamine olla probleemiks rangemate andmekaitsetingimustega klientidele. Sellisel juhul säilib siiski võimalus rakenduse eraldiseisvaks majutamiseks ühe kliendi jaoks. Tasub ka mainida, et olemasolevatele klientidele on uue rakenduse kasutuselevõtt lepingutest tulenevalt vabatahtlik.

Välimuselt on uue rakenduse kasutajaliides küllaltki sarnane vanale rakendusele (vt joonis 1 ja joonis 16). Suuremat rõhku on pandud struktuurse, mitte üks-ühele sarnasuse loomisele. Seega on sarnaselt vanale rakendusele koondunud sisendid külgribale, plokkidesse vormistatult ning ekraani põhiosas on raportid mõõdustike, jooniste ning tabelitega. Seega saab lugeda täidetuks eesmärgi realiseerida kõik vana rakenduse sisendid ning väljundid uues rakenduses. Suurimaks erinevuseks kasutajaliidestes on osad värvid, tekstisuurused ning kirjatüübid. Näiteks on vana rakenduse märkeruutude ning raadionuppude taustavärv rohekas, uuel aga must. Küll aga on uue süsteemi kasutajaliides loodud Reacti teegiga, tänu millele on vajadusel võimalik värve, suuruseid ja muid disainielemente lihtsa vaevaga muuta.



Joonis 16. STACCi kodulehel oleva näidisrakenduse seadistus uues süsteemis.

Arendusliku poole pealt erineb uus süsteem vanast märkimisväärselt. Uues süsteemis on üksteisest täielikult eraldatud sisendite töötlus ning väljundite arvutamine. Ühelt poolt lihtsustab see arendust, sest väljundi arvutamiseks on lihtne defineerida selle sõltuvused ning nende sõltuvuste väärtusi ka kasutada (vt joonis 13). Teisalt tuleb aga arendajal arvestada kõikvõimalike erinevate seadistuste ning rakenduse dünaamilisusega. Seetõttu on näiteks keerulisem leida vigu väljundite arvutamises, sest sisendite seadistused võivad erineda. Uues süsteemis on võimalik defineerida ning konfigureerida kõikide olemasolevate EEDA klientide sisendid ning väljundid. Seega on täidetud ka eesmärk, et kõik olemasolevad konfiguratsioonid on kasutatavad ühes tsentraalses rakenduses.

Mikroteenuste kasutamine lihtsustab koodibaasi loetavust ning tagab süsteemi modulaarsuse. See-eest lisab mikroteenuste ning muude taustsüsteemide nagu näiteks andmebaaside kasutamine uues rakenduses halduskoormust. Kui vana rakendus koosnes sisuliselt vaid ühest juurutatavast Streamliti rakendusest, siis uues rakenduses on eraldiseisvalt juurutatavaid teenuseid 13. Asjaolu lihtsustab küll Dockeri ning eriti Docker Compose'i kasutamine, kuid sellegi poolest vajavad teenused lisatööd. Võttes aga arvesse, et uues süsteemis ei pea mitme kliendi korral rakendusi eraldi majutama, võib kokkuvõttes taristuga tegelemiseks kuluv aeg olla samaväärne.

3.2.2. Kiirustestid

Võrdlemaks vana ning uue süsteemi jõudlust, viidi läbi kiirustestid. Täpsemalt võrreldakse keskmisi arvutusteks kulunud aegu STACCi kodulehel oleva EEDA näidisversiooni[6] väljundite peal. Katsed viiakse läbi kasutades kahte erinevat sisendite seadistust, võrreldes arvutusaegu nii optimaalse strateegia leidmisega kui ilma selleta. Mõlema rakenduse ning sisendite puhul viiakse läbi 100 katset, mille tulemusi analüüsitakse. Testis võrreldakse vaid kalkulatsiooniks, mitte kogu lehe laadimiseks kuluvat aega.

Kõik testid sooritatakse ühe ja sama masina peal. Selleks kasutatakse Dell Latitude 5420 sülearvutit 32 GB muutmälu ning Intel i5-1135G7 protsessoriga. Testimise ajal jookseb kumbki rakendus eraldiseisvalt Dockeris, kusjuures kõrvaliste protsesside arv ning ressursikasutus on sama. Testides tehakse päringuid läbi kasutajaliidese. Selle automatiseerimiseks kasutatakse Seleniumi[7] tööriista ChormeDriveriga[8].

Nagu on näha tabelist 1, võttis kodulehe seadistuse kõikide sisenditega kalkulatsioon vanal süsteemil keskmiselt 7,37 sekundit ning uuel 5,52 sekundit. Keskmiselt on seega uue süsteemi arvutusaeg 25,1% kiirem. Arvutusaegade varieeruvus praktiliselt sama. Kahe testi tulemusi võrdleva t-testi p-väärtus on <0,00001, mis on olulisusnivool α=0,05 piisav väitmaks, et uus süsteem ei ole mitte ainult sama kiire, vaid kiirem vanast süsteemist. Seeläbi on täidetud eesmärk, et uus süsteem ei tohi olla aeglasem vanast süsteemist.

Tabel 1. Keskmised kalkulatsioonide arvutusajad ning nende varieeruvused.

Seadistus	**Keskmine arvutusaeg**	**Arvutusaegade dispersioon**
Vana süsteem, kodulehe seadistus, kõik sisendid aktiveeritud	7,37 s	0,17 s^2
Uus süsteem, kodulehe seadistus, kõik sisendid aktiveeritud	5,52 s	0,17 s^2
Vana süsteem, kodulehe seadistus, optimeerimiseta	6,26 s	0,05 s^2
Uus süsteem, kodulehe seadistus, optimeerimiseta	3,94 s	0,19 s^2

[6] STACCi kodulehe EEDA näidisrakendus. https://stacc.ee/solutions/energy-cost-optimizer/.
[7] Selenium. https://www.selenium.dev/.
[8] ChromeDriver. https://sites.google.com/chromium.org/driver/.

Lisaks viidi läbi analoogsed testid nii, et sisenditest oli välja lülitatud aku sisendplokk, justkui akut energiasüsteemis ei oleks. Teisisõnu tähendab see, et kalkulatsioonide jaoks ei leita optimaalset strateegiat, sest see vajab aku olemasolu. Seeläbi langes keskmine arvutusaeg vanal süsteemil 15% ning uuel süsteemil 28,6%. Taaskord on t-testi p-väärtus üliväike ehk kiiruse vahe vana ja uue süsteemi arvutusaegade vahel on statistiliselt oluline. Optimeerimiseta on uus süsteem vanaga võrreldes keskmiselt 37,1% kiirem. Antud võrdluses on küll uue süsteemi aegade varieeruvus suurem, kuid see tuleb peamiselt ühest erindist, kus kalkuleerimiseks kulus 7,2 sekundit. Ilma erindita on uue süsteemi aegade varieeruvus 0,08.

3.2.3. Majutuskulud

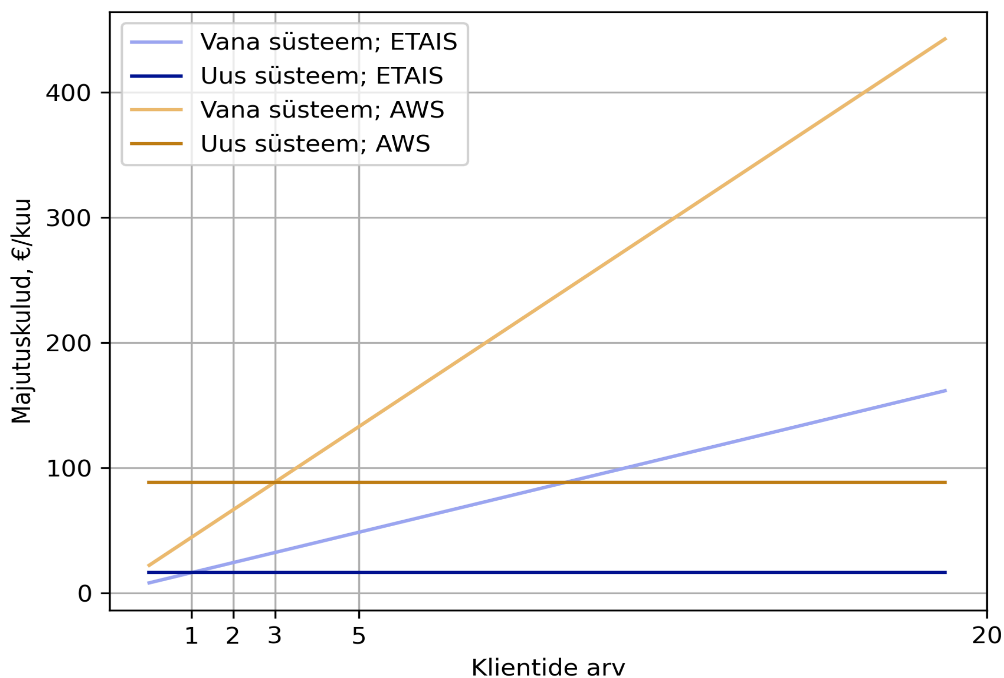
Uue ja vana süsteemi majutuskulusid saab omavahel võrrelda vaid hinnanguliselt ning seda kahel põhjusel. Esiteks tuleks vana süsteemi kogukulu leidmiseks summeerida kõikide klientide igakuised majutuskulud. Tulenevalt asjaolust, et osad kliendid on rakenduse majutamise võtnud enda IT-tiimi kanda, ei ole täpseid andmeid võimalik saada. Teiseks on vanad süsteemid majutatud peamiselt AWS-i taristul, uus süsteem on arendamise ajaks majutatud aga Eesti Teadusarvutuste Infrastruktuuri (ETAIS) serverites. Kahe teenusepakkuja hinnakirjad ning hinnastusmudelid on erinevad ning seetõttu otsest võrdlust nende kahe vahel teha ei saa.

Majutuskulude hinnanguliseks võrdlemiseks kasutatakse lihtsustatud mudelit. Nimelt eeldatakse varasema vaatluse põhjal, et vana süsteemi majutamiseks ühe kliendi jaoks läheb tarvis virtuaalmasinat 2 virtuaaltuuma (vCPU) ning 4 GB mäluga (tähistame VM1) ning et uue süsteemi majutamiseks on vaja virtuaalmasinat 4 vCPU ning 8 GB mäluga (tähistame VM2). Eeldused on realistlikud, sest samade võimsustega masinad on juba vastavate süsteemide majutamiseks kasutusel. Eeldame veel, et uus süsteem suudab etteantud parameetritega masinal teenindada kuni 20 klienti. Lihtsustatud mudelis on vana süsteemi summaarsed majutuskulud leitavad valemiga $klientide\ arv * VM1\ hind$ ning uue süsteemi majutuskulud on võrdsed VM2 hinnaga. Majutuskulusid võrreldakse nii ETAIS-e kui AWS-i hindade põhjal.

ETAIS-e hinnakirja põhjal maksab Tartu Ülikooli serverites 1 vCPU 0,003 € tunnis ning 1 GB mälu 0,0013 € tunnis[9]. Ühes kuus on 2 vCPU ja 4 GB mäluga virtuaalmasina hind seega umbes 8,08 € ning 4 vCPU ja 8GB mälu korral 16,16 €. Nagu on näha jooniselt 17, on uue süsteemi majutuskulud vanaga samaväärsed alates kolmest kliendist. Tulenevalt uue süsteemi

[9] Eesti Teadusarvutuste Infrastruktuuri hinnakiri. https://etais.ee/hinnakiri/.

majutuskulude sõltumatusest klientide arvust, on kahe süsteemi kulude vahe klientide lisandudes järjest suurem, ulatudes 20 kliendi juures 145,44 euroni.



Joonis 17. Uue ja vana süsteemi majutuskulude võrdlus ETAIS-e ning AWS-i virtuaalmasinates.

AWS-i hinnastusmudel on ETAIS-e omast keerulisem ning praktikas lisanduvad virtuaalmasinale veel mitmesugused kulud olenevalt võrguseadistustest, koormusjaoturitest ning muudest lisakomponentidest. Lihtsuse huvides võime eeldada, et kõrvalkulud on mõlemal süsteemil samad. Virtuaalmasinate hinnakirjadena kasutame *On-Demand* hinnastust Linuxi virtuaalmasinatele Stockholmi regioonis. Sobivateks virtuaalmasinateks on vana süsteemi puhul *t4g.medium* ning uue süsteemi puhul *t4g.xlarge*. Nende hinnad on vastavalt 0,03075 € tunnis ning 0,123 € tunnis[10]. Ühes kuus on masinate hinnad seega 22,14 € ning 88,56 €. Majutuskulude poolest õigustab uus süsteem end AWSi taristu peal alates neljast kliendist (vt joonis 17).

Arvestades, et hetkel on EEDA-l viis klienti, tasuks uue süsteemi juurutamine majutuskulude poolest ära nii AWS-i kui ETAIS-e taristul majutades. Seega saab edukaks lugeda ka eesmärgi, et uue süsteemi majutuskulud oleksid vana süsteemi kogukuludest väiksemad.

[10] Amazon Web Service'i virtuaalmasinate hinnakiri. https://aws.amazon.com/ec2/pricing/on-demand/.

4. Võimalikud edasiarendused

Töö raames sai energiahaldussüsteem EEDA üle viidud tarkvara teenusena kujul mikroteenustel põhinevaks veebirakenduseks. See parandas mitu varasemalt rakendusega eksisteerinud kitsaskohta ning lõi ärisuunas aluse rakenduse kui valmisteenuse müümiseks. Sellegipoolest on rakendus nii tehnoloogilises kui ka ärilises mõttes veel varajases staadiumis ning võimalikke edasiarendusi on mitmeid.

Tehnoloogiliselt tasuks analüüsida sõnumimaakleri nagu näiteks RabbitMQ[11] kasutuselevõttu. Ühelt poolt eemaldab see teenustevahelised sõltuvused, sest päringuid ei saadetaks enam otse kalkulaatori või mõne muu teenuse API otspunkti, vaid neid loetaks maakleri. Seeläbi on mitmete teenuste haldamine ning nendevahelise suhtluse loomine mõnevõrra lihtsam. Teisalt saab läbi sõnumimaakleri luua ka uut laadi funktsionaalsusi. Näiteks on üheks olemasolevaks klientide sooviks optimaalse aku suuruse soovitamine. Selle arvutamiseks kulub aga rohkem aega kui oleks mõistlik ühe API päringu jaoks ühendust lahti hoida. Sõnumimkaaleriga saaks luua toe kauajooksvate tööde teostamiseks, võimaldades jooksvalt vaadata ja uuendada töö staatust ning võtta uus töö ette alles eelmise lõpetamisel.

Rakenduse funktsionaalsustest on uuel EEDA-l hetkel puudu administratsioonikeskkond, kus saaks hallata kasutajaid ning nende konfiguratsioone. Vastavate API otspunktide näol on eeldused selleks loodud ning peamine töö seisneb kasutajaliidese loomises. Lisaks oleks rakenduse administraatoritele vajalik koondada kokku statistika rakenduse kasutamise kohta. Hetkel päringuid küll logitakse, kuid puhtalt logide põhjal võib statistika tegemine olla keeruline. Seega tuleks selle jaoks luua esmalt süsteem kasutajate tegevuste koondamiseks andmebaasi või mujale kergesti loetavasse andmehoidlasse. Seejärel saaks ära kasutada loodud seireteenuse Grafana tööriista vastavate näidikulaudade loomiseks.

EEDA on hetkel suunatud peamiselt äriklientidele ehk ettevõtetele, kes soovivad tasuvusarvutusi oma klientidele teostada. Ärikliendid kasutavad EEDA-t regulaarselt, neil on selleks oma kindel konfiguratsioon ning on nõus maksma selle eest igakuist tasu. Küll aga oleks äriliselt võimalik rakendust kohandada ka eraklientidele, kes soovivad tasuvusarvutust teostada ühekordselt oma majapidamise jaoks. Sellisel juhul tuleks luua uus hinnastusmudel, mis arvutab rakenduse kasutamise tasu jooksvalt vastavalt päringutele ning nende mahule.

[11] RabbitMQ. https://www.rabbitmq.com/.

Kokkuvõte

Töö üldine eesmärk oli energiahaldussüsteemi EEDA viimine monoliitselt ahiritektuurilt ning kliendispetsiifiliselt lähenemiselt mikroteenustel põhinevale arhitektuurile ning tarkvara teenusena mudelile. Ärilise poole pealt loob see eeldused rakenduse kui valmisteenuse müümiseks, mis vähendab süsteemi majutuskulusid ning võimaldab klientidel kiiremini rakendust kasutama hakata. Mikroteenuste kasutamine muudab tänu funktsionaalsuste eraldamisele koodibaasi hallatavamaks ning lihtsustab seeläbi arendusprotsessi. Seati neli konkreetsemat eesmärki, mille täitmise korral saab töö edukaks lugeda: 1) kõikide olemasolevate klientide seadistusi on võimalik üle viia uude süsteemi; 2) kasutajaliideses on säilinud kõik sisendid ning väljundid; 3) uue rakenduse majutuskulud on väiksemad; 4) uus rakendus ei ole arvutuskiiruste poolest aeglasem.

Süsteemiarhitektuuris eraldati EEDA kolmeks tuumsüsteemi teenuseks: kalkulaator, optimeerija ning päikesepaneelide tootlikkuse ennustusteenus. Kalkulaator vastutab kliendispetsiifiliste konfiguratsoonide haldamise, kasutaja sisendite vastuvõtmise ning väljundite arvutamise eest. Väljundite jaoks loodi muuhulgas sõltuvustel põhinev loogika, mis võimaldab neid koodis seadistada ja arvutada erinevate sisendite korral. Kalkulaatori teenuses on seadistavad kõik olemasolevate EEDA klientide sisendite ning väljundite konfiguratsioonid, täites sellega esimese neljast töö eesmärgist. Kalkulaator kasutab arvutuste läbiviimiseks nii PV-ennustuste kui ka optimeerija teenust. Tänu teenuste eraldiseisvusele võimaldab see tulevikus nende funktsionaalsuse liidestamist ka kõrvaliste teenustega.

Uuele rakendusele loodi äriloogikast eraldatud kasutajaliides, mis läbi rakendusliidese kalkulaatoriga suhtleb. See võimaldas uus rakendus luua visuaalselt sarnane vanale rakendusele. Kasutajaliideses on realiseeritud kõik varasema rakenduse funktsionaalsused, mis täidab töö teise eesmärgi. Lisaks on süsteemis võetud kasutusele tugiteenused autentimiseks, seireks ning masinõppemudelite halduseks. Autentimisteenus tegeleb nii kasutajate autentimise kui ka õiguste haldamisega, tagades rakenduse turvalisuse. Seireteenus suurendab süsteemi töökindlust läbi arvutusressursside kohta käivate andmete kogumise, visualiseerimise ning automaatteadete saatmise. Mudelihaldus eraldab ülejäänud teenuste loogikast masinõppespetsiifika, parandades seeläbi süsteemi hallatavust.

Uues rakenduses on kalkulatsioonid võrreldes vana rakendusega kiiremad. STACC-i kodulehel oleva näidisrakenduse seadistuste korral paranesid arvutusajad kõikide sisendite korral keskmiselt 25,1% ning akusisenditeta arvutamise korral 37,1%. Lisaks on uue süsteemi

majutuskulud väiksemad olemasolevate EEDA süsteemide kogukuludest. Seega on täidetud ka töö kolmas ning neljas eesmärk.

Viidatud kirjandus

[1] Masso M. Aktiivsed tarbijad tuleviku energiasüsteemis. Arengusuundumused aastani 2040. Arenguseire Keskus. 2024. https://arenguseire.ee/raportid/aktiivsed-tarbijad-tuleviku-energiasusteemis-arengusuundumused-aastani-2040/ (28.03.2025).

[2] Mišljenović N, Šimić Z, Topić D, Knežević G. An algorithm for the optimal sizing of the PV system for prosumers based on economic indicators and the input data time step. Solar Energy 2023; 262:111882. https://doi.org/10.1016/j.solener.2023.111882.

[3] STACC - Energeetika. https://stacc.ee/et/energeetika/ (20.04.2025).

[4] Streamlit. https://streamlit.io/ (20.04.2025).

[5] Streamlit Data Flow. https://docs.streamlit.io/get-started/fundamentals/main-concepts#data-flow (30.11.2024).

[6] Streamlit Community Cloud. https://streamlit.io/cloud (30.11.2024).

[7] Anderson R, Nguyen R, Duarte N, Sylvester-Ogan T, Mahmoud M. The suitability of software as a service (SAAS) as the replacement to traditional software. 2022 International Conference on Computational Science and Computational Intelligence (CSCI), IEEE; 2022, lk 1931–5. https://doi.org/10.1109/CSCI58124.2022.00348.

[8] Ouh EL, Kok Siew Gan B. An Exploratory Study of Architectural Style and Effort Estimation for Multi-Tenant Microservices-Based Software as a Service (SaaS). 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), IEEE; 2023, lk 159–66. https://doi.org/10.1109/ICSA-C57050.2023.00043.

[9] Microsoft 365. https://www.microsoft.com/et-ee/microsoft-365/microsoft-office (07.12.2024).

[10] Shapouri F, Ward K, Setor T. Determinants of software as a service (saas) adoption. Journal of Computer Information Systems 2024; 64:301–13. https://doi.org/10.1080/08874417.2023.2199270.

[11] Kapikul A, Savić D, Milić M, Antović I. Application development from monolithic to microservice architecture. 2024 28th International Conference on Information Technology (IT), IEEE; 2024, lk 1–4. https://doi.org/10.1109/IT61232.2024.10475769.

[12] Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. I trükk. Beijing: O'Reilly Media. 2019.

[13] Gos K, Zabierowski W. The comparison of microservice and monolithic architecture. 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH), IEEE; 2020, lk 150–3. https://doi.org/10.1109/MEMSTECH49584.2020.9109514.

[14] Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, et al. Microservices: yesterday, today, and tomorrow. ArXiv 2016. https://doi.org/10.48550/arxiv.1606.04036.

[15] Abgaz Y, McCarren A, Elger P, Solan D, Lapuz N, Bivol M, et al. Decomposition of monolith applications into microservices architectures: A systematic review. IIEEE Trans Software Eng 2023; 49:4213–42. https://doi.org/10.1109/TSE.2023.3287297.

[16] Pathak G, Singh M. A review of cloud microservices architecture for modern applications. 2023 World Conference on Communication & Computing (WCONF), IEEE; 2023, lk 1–7. https://doi.org/10.1109/WCONF58270.2023.10235199.
[17] Kazanavičius J, Mažeika D. An Approach to Migrate from Legacy Monolithic Application into Microservice Architecture. 2023 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream), IEEE; 2023, lk 1–6. https://doi.org/10.1109/eStream59056.2023.10135021.
[18] Berry V, Castelltort A, Lange B, Teriihoania J, Tibermacine C, Trubiani C. Is it worth migrating a monolith to microservices? An experience report on performance, availability and energy usage. 2024 IEEE International Conference on Web Services (ICWS), IEEE; 2024, lk 944–54. https://doi.org/10.1109/ICWS62655.2024.00112.
[19] Google Trends: Cloud Computing. https://trends.google.com/trends/explore?date=all&q=cloud%20computing&hl=en (08.12.2024).
[20] Mell P, Grance T. The NIST Definition of Cloud Computing. National Institute of Standards and Technology Special Publication 2011; 800–145.
[21] Sharma S. Evolution of as-a-Service Era in Cloud. ArXiv 2015. https://doi.org/10.48550/arxiv.1507.00939.
[22] Williams D, Garcia J, Crosby S. An introduction to virtualization. Virtualization with Xen™, Elsevier; 2007, lk 1–42. https://doi.org/10.1016/B978-159749167-9/50006-0.
[23] Nanda S, Chiueh T. A. Survey on Virtualization Technologies 2005.
[24] Amankwah R, Asianoa R, Birago B. Virtualization and cloud computing. IJCA 2020; 176:1–5. https://doi.org/10.5120/ijca2020920418.
[25] Bentaleb O, Belloum ASZ, Sebaa A, El-Maouhab A. Containerization technologies: taxonomies, applications and challenges. J Supercomput 2022; 78:1144–81. https://doi.org/10.1007/s11227-021-03914-1.
[26] Watada J, Roy A, Kadikar R, Pham H, Xu B. Emerging trends, techniques and open issues of containerization: A review. IEEE Access 2019; 7:152443–72. https://doi.org/10.1109/ACCESS.2019.2945930.
[27] Mavridis I, Karatza H. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. Future Generation Computer Systems 2019; 94:674–96. https://doi.org/10.1016/j.future.2018.12.035.
[28] Pahl C, Brogi A, Soldani J, Jamshidi P. Cloud Container Technologies: A State-of-the-Art Review. IEEE Trans Cloud Comput 2019; 7:677–92. https://doi.org/10.1109/TCC.2017.2702586.
[29] Haque A, Rahman R, Rahman S. Microservice-based Architecture of a Software as a Service (SaaS) Building Energy Management Platform. 2020 6th IEEE International Energy Conference (ENERGYCon), IEEE; 2020, lk 967–72. https://doi.org/10.1109/ENERGYCon48941.2020.9236617.
[30] Jones S, Charlesworth R, Naik K, Charlesworth T, O'Dwyer E, Ianakiev A, et al. A multi-energy system optimisation software for advanced process control using hypernetworks and a micro-service architecture. Energy Reports 2021; 7:167–75. https://doi.org/10.1016/j.egyr.2021.08.159.
[31] Johnson J. Hypernetworks for reconstructing the dynamics of multilevel systems,

Milton Keynes, UK: Design-Complexity Group, The Open University; 2006.
[32] Yudha Erian Saputra Moch, Noprianto, Noor Arief S, Nur Wijayaningrum V, Syaifudin YW. Real-Time Server Monitoring and Notification System with Prometheus, Grafana, and Telegram Integration. 2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETSIS), IEEE; 2024, lk 1808–13. https://doi.org/10.1109/ICETSIS61505.2024.10459488.
[33] Chen A, Chow A, Davidson A, DCunha A, Ghodsi A, Hong SA, et al. Developments in mlflow: A system to accelerate the machine learning lifecycle. Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning, New York, NY, USA: ACM; 2020, lk. 1–4. https://doi.org/10.1145/3399579.3399867.
[34] Megargel A, Poskitt CM, Shankararaman V. Microservices orchestration vs. choreography: A decision framework. 2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC), IEEE; 2021, lk 134–41. https://doi.org/10.1109/EDOC52215.2021.00024.
[35] Kane S, Matthias K. Docker: Up & Running: Shipping Reliable Containers in Production. III trükk. O'Reilly Media; 2023.
[36] What is Docker?. https://docs.docker.com/get-started/docker-overview/ (17.04.2025).
[37] Docker Hub. https://www.docker.com/products/docker-hub/ (20.04.2025).
[38] Kithulwatta WMCJT, Jayasena KPN, Kumara BTGS, Rathnayaka RMKT. Performance Evaluation of Docker-based Apache and Nginx Web Server. 2022 3rd International Conference for Emerging Technology (INCET), IEEE; 2022, lk 1–6. https://doi.org/10.1109/INCET54531.2022.9824303.
[39] Docker Compose. https://docs.docker.com/compose/ (20.04.2025).
[40] Nginx. https://nginx.org/en/ (20.042025).
[41] Netcraft March 2025 Web Server Survey. https://www.netcraft.com/blog/march-2025-web-server-survey/ (20.04.2025).
[42] Kte'pi B. Python (programming language). Salem Press Encyclopedia of Science.
[43] TIOBE Index. https://www.tiobe.com/tiobe-index/ (07.04.2025).
[44] PyPI - The Python Package Index. https://pypi.org/ (20.04.2025).
[45] Butwall M, Ranka P, Shah S. Python in field of data science: A review. IJCA 2019; 178:20–4. https://doi.org/10.5120/ijca2019919404.
[46] FastAPI. https://fastapi.tiangolo.com/ (20.04.2025).
[47] FastAPI Alternatives, Inspiration and Comparisons. https://fastapi.tiangolo.com/alternatives/ (20.04.2025).
[48] 2024 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/technology#2-web-frameworks-and-technologies (20.04.2025).
[49] TechEmpower Framework Benchmarks. https://www.techempower.com/benchmarks/#section=test&runid=7464e520-0dc2-473d-bd34-dbdfd7e85911&hw=ph&test=query&l=hra0hr-cn2 (20.04.2025).
[50] Banks A, Porcello E. Learning React: Modern Patterns for Developing React Apps. II trükk. O'Reilly Media. 2020.
[51] Statista: Most used web frameworks among developers 2024.

https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/ (15.04.2025).
[52] React. https://react.dev/ (15.04.2025).
[53] Next.js. https://nextjs.org/docs (20.04.2025).
[54] Next.js Routing: Middleware. https://nextjs.org/docs/app/building-your-application/routing/middleware (20.04.2025).
[55] Hoops F, Matthes F. A Middleware Architecture for Self-Sovereign Identity Authentication and Authorization. 2024 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS), IEEE; 2024, lk 79–85. https://doi.org/10.1109/DAPPS61106.2024.00019.
[56] cAdvisor. https://github.com/google/cadvisor (20.04.2025).
[57] cAdvisor: Storage. https://github.com/google/cadvisor/tree/master/docs/storage (20.04.2025).
[58] Prometheus Overview. https://prometheus.io/docs/introduction/overview/ (20.04.2025).
[59] About Grafana. https://grafana.com/docs/grafana/latest/introduction/ (20.04.2025).
[60] MLflow. https://mlflow.org/docs/latest/ (20.04.2025).
[61] FastAPI Users. https://fastapi-users.github.io/fastapi-users/latest/ (20.04.2025).
[62] Django | Working with forms. https://docs.djangoproject.com/en/5.2/topics/forms/ (14.05.2025).
[63] Django | Form fields. https://docs.djangoproject.com/en/5.2/ref/forms/fields/ (14.05.2025).
[64] Python documentation | Metaclasses. https://docs.python.org/3/reference/datamodel.html#metaclasses (14.05.2025).
[65] Esterbrook C. Using Mix-ins with Python 2001. https://www.linuxjournal.com/article/4540 (14.05.2025).
[66] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. III trükk. Cambridge, Mass: MIT Press. 2009.
[67] Fowler M. Patterns of Enterprise Application Architecture. I trükk. Boston: Addison-Wesley Professional. 2002.
[68] Percival H, Gregory B. Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. I trükk. Beijing: O’Reilly Media. 2020.

Lisad

Lisa 1. Sisendplokkide näited

Battery

Battery

Optimization type

Automatic

Manual

Battery cost [EUR]

300000

Battery change cost [%]

20

Nominal battery capacity [kWh]

1900

Battery cycles

7000

Battery cycles in a day limit

2.0

Depth of Discharge [%]

90

Battery efficiency [%]

90

Battery degradation [%/year]

2

Battery usage threshold setting

Automatic

Manual

Battery manual usage threshold [EUR/MWh]

Inverter power [kW]

500

Battery

Battery investment [EUR]

5000

Battery capacity [kWh]

9.6

Battery C-rate

0.50

0.10 1.00

Battery cycles

7000

Battery charge range [%]

5 95

0 100

Inverter power

8

Joonis 1.1. Näide kahest erinevast aku sisendploki seadistusest.

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Karl Kevin Ruul,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **Energiahaldussüsteem pilvepõhise teenusena: arhitektuuriline ja arenduslik käsitlus**, mille juhendajad on Tiit Sepp ja Kirill Grjaznov, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada Tartu Ülikooli digitaalarhiivi kuni autoriõiguse kehtivuse lõppemiseni;
2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni;
3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile;
4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Karl Kevin Ruul

15.05.2025