Topi Kanninen

SUORITUSKYKYTESTAUKSEN TOTEUTTAMINEN JA SUORITUSKYKYBUDJETIN MÄÄRITTÄMINEN

CASE FINAZILLA OY

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2024

TIIVISTELMÄ

Kanninen, Topi
Suorituskykytestauksen toteuttaminen ja suorituskykybudjetin määrittäminen
Case Finazilla Oy
Jyväskylä: Jyväskylän yliopisto, 2024, 60 s.
Tietojärjestelmätiede pro gradu -tutkielma
Ohjaaja: Seppänen, Ville

Ohjelmiston suorituskyky on merkittävä tekijä nykyaikaisissa tietojärjestelmissä, jotka joutuvat käsittelemään alati kasvavia datamääriä. Tässä tutkielmassa tutkittiin, miten tätä ohjelmiston suorituskykyä voidaan testauttaa sekä valvoa suorituskykybudjetin avulla. Tutkielman tavoitteena oli luoda tutkimuksen case-yrityksen tarpeisiin ohjelmiston suorituskykyä testaavat suorituskykytestit sekä kehittää suorituskykybudjetti, minkä avulla ohjelmiston suorituskyvyn kehittymistä pystyttäisiin muutosten alla valvomaan. Tutkielmassa käsiteltiin ohjelmistotestaukseen ja erityisesti suorituskykytestaukseen liittyviä keskeisiä käsitteitä. Tunnistamalla oikeat testityypit ja -tasot, voitiin testaus kohdistaa haluttuihin osiin ohjelmistosta ja näin saavuttaa tavoiteltu testauspalaute. Tutkielmassa kartoitettiin myös suorituskykybudjetin laadintaan sekä sisältöön vaikuttavia tekijöitä. Suorituskykybudjetin kohdalla on tärkeää tunnistaa oikeat tavoitteet, joihin ohjelmiston toimintaa ja suorituskykyä halutaan viedä. Näiden päätösten pohjalta on mahdollista rakentaa tavoitteita tukevaa suorituskyvyn budjetointia. Tutkielmassa sovellettiin suunnittelutieteellistä tutkimusmenetelmää, minkä avulla kehitettiin suorituskyvyn testaukseen sekä budjetointiin soveltuvia ratkaisuja case-yritykselle. Tutkielman tuloksena syntyi suorituskyvyn testauksen sekä valvonnan prosessi, mikä integroitiin osaksi case-yrityksen tuotekehitystä.

Asiasanat: ohjelmistotestaus, suorituskykytestaus, suorituskykybudjetti, automaatiotestaus

ABSTRACT

Kanninen, Topi
Implementation of performance testing and definition of performance budget
Case Finazilla Oy
Jyväskylä: University of Jyväskylä, 2024, 60 pp.
Information Systems Science Master's Thesis
Supervisor: Seppänen, Ville

Software performance is a major factor in modern information systems that have to handle ever-increasing amounts of data. This thesis investigated how this software performance can be tested and monitored using a performance budget. The aim of the thesis was to create performance tests to test the performance of the software for the needs of the case study company and to develop a performance budget to monitor the evolution of the software performance in the face of change. The study covered the key concepts related to software testing and in particular performance testing. By identifying the right test types and levels, testing could be targeted at the desired parts of the software to achieve the desired testing feedback. The study also identified the factors influencing the preparation and content of the performance budget. For the performance budget, it is important to identify the right objectives to which the software's performance and performance are to be driven. Based on these decisions, it is possible to build a performance budget that supports the objectives. The thesis applied a design science research methodology to develop performance testing and budgeting solutions for a case study company. As a result, a performance testing and monitoring process was developed and integrated into the product development of the case company.

Keywords: software testing, performance testing, performance budget, test automation

KUVIOT

KUVAT

TAULUKOT

SISÄLLYS

1 Johdanto

Ohjelmistotestaus on tärkeä osa-alue ohjelmiston kehityksen elinkaarta, mutta samalla osa-alue, joka jää usein liian vähäiselle huomiolle. Riskit ohjelmiston testauksen sivuuttamiselle voivat sisältää esimerkiksi seuraavia vaikutuksia kehitettävään ohjelmistoon:

- Ohjelmiston huono laatu
- Tietoturvaongelmat
- Kasvavat ylläpitokulut
- Huonot käyttäjäkokemukset
- Suorituskyvyn heikkeneminen

Tästä syystä ohjelmistotestauksen roolia osana ohjelmiston tai järjestelmän elinkaarta ei voida väheksyä tai sivuuttaa. (*The Risks of Performing No Testing or Minimal Testing*, n.d.) Myös Neves ym. (2024) tunnistavat tutkimuksessaan ohjelmistotestauksen suuren roolin osana ohjelmiston elinkaarta, kun halutaan varmistua siitä, että kehitettävä ohjelmisto vastaa sen tulevien käyttäjien vaatimuksia. Tällöin testausprosessin tulisi sisältää validointeja ohjelmiston ominaisuuksista, käyttäytymisestä sekä laadusta. Boukhlif ym. (2023) vastaavasti tunnistavat artikkelissaan, että ohjelmistojen alati nopeutuva kehityssykli sekä samalla monimutkaisten ominaisuuksien kasvu aiheuttavat sen, että ohjelmistojen laadun takaaminen on alati vaikeampaa. Ja koska hyvin toteutettu ohjelmistotestaus auttaa kehittäjiä havaitsemaan ohjelmiston virheet sekä puutteet jo kehityskaareen varhaisimmissa vaiheissa, voidaan testaus nostaa yhdeksi tärkeimmistä ohjelmistokehityksen vaiheista.

Tämän tutkielman aiheena on suorituskykytestauksen sekä suorituskykybudjetin määrittäminen tutkimuksen case-yritykselle. Tutkimuksen tutkimuskysymyksenä on ''Miten suorituskykytestaus voidaan toteuttaa ja budjetoida?''. Tutkimuskysymys voidaan jakaa myös kahteen eri tutkimuskysymykseen seuraavasti:

- Miten suorituskykyä voidaan budjetoida?
- Miten suorituskykytestaus voidaan toteuttaa?

Tutkimuksessa teoriaosuus keskittyy käsittelemään ohjelmistotestausta, suorituskykytestausta sekä suorituskykybudjettia käsitteinä, sekä niiden soveltumista kohdeyrityksenyrityksen kontekstiin. Tutkimuksen tavoitteena on löytää validoinnin pohjalta sopivin teknologia suorituskykytestauksen toteuttamiselle, sekä laatia testien ohelle suorituskykybudjetti, jolla kohdeyrityksen ohjelmiston suorituskykyä voidaan testejä hyödyntämällä valvoa.

1.1 Kohdeyritys

Kohdeyrityksenä tutkimukselle toimii Finazilla Oy. Finazilla Oy on Jyväskylässä vuonna 2018 perustettu konsulttitalo, joka tarjoaa asiakkailleen ratkaisuja talousjohtamiseen erilaisten liiketoiminnan analyysi-, budjetointi-, raportoin- sekä ennustamistyökalujen muodossa. (Finazilla, n.d.) Finazilla Oy on osa Lemonsoft-konsernia. Lemonsoft Oyj on vuonna 2006 Vaasassa perustettu suomalainen ohjelmistoyritys, joka tarjoaa asiakkailleen erilaisia ERP-ratkaisuja. Lemonsoft tarjoaa ratkaisujaan erityisesti pienille sekä keskisuurille yrityksille muun muassa taloushallinnon, tuotannonohjauksen, logistiikan sekä asiakkuudenhallintaan liittyvien tuotteiden avulla. Vuonna 2022 konserni työllisti noin 200 henkilöä ja palveli yli 7000 asiakasta. (Lemonsoft, n.d.)

Ohjelmistojen suorituskyky on yhä tärkeämmässä roolissa nykyaikaisissa tietojärjestelmissä. Suorituskykytestaus on osa-alue, jolla voidaan varmistaa ohjelmistojen toiminnan tehokkuus suurien käyttäjä- sekä kuormitusmäärien alla. Sommerville (2011) luokittelee ohjelmiston suorituskyvyn yhdeksi ohjelmistosuunnittelun perusteista. Hänen mukaansa suorituskyky on kriittinen osa mitä tahansa ohjelmistoa. Ohjelmiston tulisi käyttäytyä virheettömästi sille odotetulla tavalla sekä olla aina käytettävissä. Ohjelmiston tulisi myös olla toiminnoiltaan turvallinen sekä tehokas, eikä sen tulisi sallia tuhlata käytettävissä olevia resursseja.

Ohjelmistojen resurssien oikea sekä tehokas allokointi erilaisten muutosten alla liittyy vahvasti suorituskyvyn ylläpitoon, ja tähän suorituskykybudjetti on tärkeä sekä valvonta- että ohjaustyökalu. Case-yrityksen kaltaisten ohjelmistotalojen tapauksessa huonosti hallittu ja seurattu suorituskyky voi johtaa kriittisiin toimintahäiriöihin, jotka taas vaikuttavat osaltaan yrityksen asiakkaiden omiin liiketoimintoihin. Tästä syystä ohjelmiston suorituskyvyn optimointi sekä budjetointi ovat tärkeitä osa-alueita muun muassa case-yrityksen kasvun ja kilpailukyvyn kannalta.

Tutkimuksesta saavutettavia käytännön hyötyjä ovat erityisesti ratkaisut automatisoidumpaan suorituskyvyn ylläpitoon sekä parantamiseen osana nykyisiä tuotekehitysprosesseja. Suorituskykytestauksella on mahdollista tunnistaa erilaisia järjestelmän toimintaa hidastavia pullonkauloja, sekä optimoida järjestelmän resurssien käyttöä, mikä nostaa järjestelmän kykyä vastata käyttäjien erilaisiin tarpeisiin. Tämän lisäksi erillisen suorituskykybudjetin määrittäminen antaa kehitystiimille selkeämmät

suorituskyvyn tavoitteet sekä rajoitukset, joiden avulla pystytään ehkäisemään ohjelmiston suorituskyvyn heikentyminen paremmin. Näinollen tässä tutkimuksessa voidaan tuottaa case-yrityksen tuotekehitysprosessia konkreettisesti tukevia ratkaisuja.

Tutkimuksessa uuden tiedon tuottaminen keskittyy erityisesti suorituskykybudjetin käsitteen ympärille. Kun ohjelmisto- sekä suorituskykytestausta on tutkittu paljon, on suorituskykybudjetti varsin tuntematon mutta sitäkin tärkeämpi ohjelmistokehityksen työkalu. Suorituskykybudjettia hyödyntämällä voidaan luoda konkreettisia resursseihin sekä vasteaikoihin liittyviä mittareita, jotka pohjautuvat testattavaan järjestelmään, sekä sen käyttöympäristöihin että -tarkoituksiin. Tällä tutkimuksella voidaan tarjota teoriapohjaa sekä käytännön esimerkkejä suorituskykybudjetin laadintaan, joita voidaan hyödyntää vertailukohtana sekä näkökulmana tulevaisuudessa eri suorituskykybudjettien laadinnassa.

Tutkielman rakenne etenee seuraavasti. Johdannossa esitellään tämän tutkielman aihe sekä tutkimuskysymykset. Tutkielman toisessa luvussa käydään läpi keskeisiä käsitteitä liittyen ohjelmistotestaukseen sekä testausautomaatioon. Luvun tavoitteena on määrittää käsitteistöä, jonka pohjalta testausta voidaan kohdistaa oikein halutun testaustuloksen saavuttamiseksi. Kolmannessa luvussa käydään tarkemmin läpi suorituskykytestaukseen liittyvää käsitteistöä, joiden avulla voidaan paremmin tunnistaa suorituskykytestauksen oikeat tyypit sekä niiden tavoitteet. Luvussa käydään läpi myös suorituskyvyn pullonkauloja, joita suorituskykytestauksella pyritään havaitsemaan. Näiden kahden edellisen luvun pohjalta pyritään tutkielmassa vastaamaan tutkimuskysymykseen suorituskykytestauksen toteuttamisesta. Tutkielman neljännessä luvussa toteutetaan kirjallisuuskatsaus, minkä tavoitteena on kartoittaa suorituskykybudjettia käsitteenä sekä niitä keskeisiä käsitteitä, joita suorituskykybudjetin käyttöön sekä rakentamiseen kuuluu. Tämän luvun pohjalta vastataan tutkielman toiseen tutkimuskysymykseen suorituskykybudjetin määrittelemisestä case-yritykselle. Viidennessä luvussa käydään läpi suunnittelutieteellistä tutkimusmenetelmää sekä kirjallisuuskatsausta, jotka valikoituivat tutkielmassa käytettäviksi tutkimusmenetelmiksi. Kuudennessa luvussa luodaan suunnittelutieteellisen tutkimusmenetelmää hyödyntäen artefakti, jonka avulla pyritän vastaamaan tutkielman tutkimuskysymyksiin. Kappaleessa kuvataan myös artefaktin käyttöönotto osaksi case-yrityksen tuotekehitystä. Seitsemännessä luvussa pohditaan tutkielman onnistumista tieteellisestä sekä käytännön toteutusten näkökulmasta, pohditaan tutkielman mahdollisia rajoitteita sekä avataan jatkotutkimusaiheita. Kahdeksannessa luvussa vedetään yhteen tutkielman tulokset sekä sen merkitys case-yritykselle.

2 OHJELMISTOTESTAUS

Tässä kappaleessa käsitellään ohjelmistotestausta sekä määritellään siihen kuuluvaa termistöä. Kappaleessa käydään läpi ohjelmistotestauksen eri tasoja sekä tyyppejä, jotka määrittävät osaltaan sen, millaisella laadulla ja laajuudella ohjelmistoja voidaan testata. Lisäksi kappaleessa käsitellään testaamisen automatisointia.

Ohjelmistotestausta kuvataan joukoksi eri käytänteitä sekä prosesseja, joiden tarkoituksena on tarkistaa, vastaavatko ohjelmiston todelliset tulokset sille asetettuja vaatimuksia, sekä varmistaa, että ohjelmisto toimii virheettömästi. Ohjelmistotestauksen tavoitteena on tunnistaa ohjelmiston erilaiset virheet sekä puutteet. Ohjelmistotestausta voidaan suorittaa sekä manuaalisesti että automatisoituna erilaisten skriptien sekä työkalujen avulla. (Hourani ym., 2019)

Ohjelmistotestauksen tavoitteena on myös todentaa, että ohjelmisto täyttää sille asetetut liiketoiminnalliset sekä tekniset vaatimukset, mitkä ovat ohjanneet ohjelmiston suunnittelua sekä kehittämistä. Ohjelmistotestausta käytetään myös testaamaan muita ohjelmiston laatutekijöitä, kuten esimerkiksi luotettavuutta, eheyttä, turvallisuutta sekä tehokkuutta. Testauksen lähestymistapa on hyvin paljon riippuvainen siitä, millaista ohjelmistoa testataan, millaisella tasolla ja millä tarkoitusperillä. (S.M.K & Farooq, 2010)

International Software Testing Qualifications Board ISTQB (2023) on määritellyt ohjelmistotestaukselle erilliset testaustasot sekä -tyypit luokittelemaan testien käyttötarkoituksia. Testaustasot ovat testaustoimintojen ryhmiä, joita organisoidaan sekä hallitaan yhdessä. Kukin testitaso on testausprosessin osa, joka suoritetaan tietyssä vaiheessa tuotekehitysprosessia aina yksittäisestä komponentista kokonaisien järjestelmien testaukseen. Testaustasot ovat liitännäisiä muihin toimintoihin sovelluksen kehitysprosessissa, eli ”Software Development Life Cycle” SDLC:ssä. Vaiheittaisissa SDLC-malleissa testitasot määritellään usein niin, että yhden tason poistumiskriteerit ovat osa seuraavan tason sisäänmenokriteerejä. Joissakin iteratiivisissa malleissa tämä ei välttämättä kuitenkaan päde, sillä kehitysprosessin vaiheet voivat ulottua useiden eri testitasojen läpi. Eri testitasot voivat olla myös ajallisesti päällekkäisiä. Testiryhmät ovat vastaavasti testitoimintojen ryhmiä, jotka liittyvät toisiinsa jonkin tietyn laatuominaisuuden

kautta. Useimmat saman ryhmän testitoiminnot voidaan suorittaa kaikilla eri testitasoilla.

2.1 Testaustasot

Tässä luvussa käydään läpi ohjelmistotestauksen eri tasoja, sekä sitä, miten eri testaustasot eroavat toisistaan testauksen näkökulmasta. Eri testaustasot erotetaan ISTQB:n (2023) mukaan toisistaan seuraavalla, taulukossa 1 esiteltävällä ei-tyhjentävällä määrittelyllä, jonka tavoitteena on välttää testaustoimintojen päällekkäisyys. Testaustasoja erottavia tekijöitä ovat testauksen kohde, tavoitteet, testauksen perusta, etsittävät viat ja virheet, sekä testauksessa käytettävä lähestymistapa että testausvastuun jakaminen.

Testauksen kohteella tarkoitetaan sitä järjestelmän osaa tai komponenttia, minkä käyttäytymistä testauksella tarkastellaan. Testaustason kohteena voi olla joko yksittäinen komponentti, järjestelmän kokonaisuuden toiminnan tarkasteleminen tai käyttäjän tekemät toiminnot, joilla pyritään varmistamaan järjestelmän oikea toimivuus loppukäyttäjien käytössä.

Testaukselle asetetut tavoitteet erottelevat myös eri testaustasot toisistaan. Testauksen tavoitteet voivat olla esimerkiksi järjestelmän komponenttitasolla, jolloin tavoitteena on varmistaa yksittäisen komponentin toimivuus. Vastaavasti hyväksymistestauksessa tavoitteena on varmistaa, että järjestelmä täyttää sille asetetut liiketoimintavaatimukset sekä käyttäjien eri tarpeet. Testauksen perustalla tarkoitetaan sitä, mihin testauksen suunnittelu sekä toteutus perustuu. Perustana voivat toimia esimerkiksi määrittelydokumentit, suunnitteluvaatimukset tai käyttäjävaatimukset.

Testaustason tavoitteena havaittavat järjestelmän viat ja virheet erottelevat myös osaltaan eri testaustasot toisistaan. Järjestelmätasolla tavoitteena on havaita virheitä, jotka estävät järjestelmän eri komponenttien yhteistoiminnan. Komponenttitasolla taas keskitytään löytämään yksittäisten komponenttien toimintoja estäviä virheitä.

Viimeiseksi testauksen lähestymistapa sekä vastuut erottelevat testaustasot toisistaan. Testauksen lähestymistapa määrittelee, miten testaus varsinaisesti suoritetaan. Testit voidaan esimerkiksi suorittaa joko manuaalisesti tai automatiikan avulla. Testauksen vastuut sekä vastuuhenkilöt vaihtelevat myös testaustason mukaan: yksittäisen komponentin testaus on yleensä kehittäjän vastuulla, kun vastaavasti hyväksymistestauksen suorittaa tyypillisesti asiakas tai loppukäyttäjä

Taulukko 1. Eri testaustasojen erottavat tekijät

Erottava tekijä	**Selite**
Kohde	- Yksittäinen komponentti - Kokonaisjärjestelmä - Käyttäjien toiminnot
Tavoitteet	- Yksittäisen komponentin toimivuus - Järjestelmän osien yhteistoiminnan varmistaminen - Liiketoiminnallisten vaatimusten ja käyttäjien tarpeiden täyttäminen
Perusta	- Määrittelydokumentit - Suunnitteluvaatimukset - Käyttäjävaatimukset - Liiketoimintaprosessit
Viat ja virheet	- Komponentin virheet - Järjestelmän virheet - Käyttäjän vaatimuksia vastaamattomat toiminnallisuudet
Lähestymistapa ja vastuut	- Manuaalinen vs. automaatiotestaus - White-Box vs. Black-box-testaus - Yksikkötestaus -> kehittäjä - Integraatiotestaus -> testausryhmä - Hyväksymistestaus -> loppukäyttäjä

Erilaisiksi testaustasoiksi on määritelty yksikkötestaukset, yksikköintegraatiotestaukset, järjestelmätestaukset, järjestelmän integraatiotestaukset sekä hyväksymistestaukset. (ISTQB, 2023)

Yksikkötestauksessa, eli komponenttitestauksessa, keskitytään yksittäisten komponenttien testaamiseen eristetyssä ympäristössä. Yksikkötestaus vaatii usein toteutuakseen erityistä tukea, kuten esimerkiksi yksikkötesteille rakennettuja testikehyksiä. Testaus suoritetaan yleensä sovelluksen kehittäjien toimesta heidän omissa kehitysympäristöissään. (ISTQB, 2023) Yksikkötestauksen hyötynä voidaan pitää sitä, että se mahdollistaa erikseen yksittäisten komponenttien testaamisen erikseen, ja mahdollisten virheiden löytämisen aikaisessa vaiheessa (Huizinga & Kolawa, 2007).

Yksikköjen integraatiotestauksen tavoitteena on määrittää, toimivatko erikseen rakennetut ja toteutetut komponentit oikein, kun ne yhdistetään toisiinsa (Fowler, 2018). Testauksessa keskitytään komponenttien välisten rajapintojen sekä niiden vuorovaikutusten testaamiseen. Komponenttien integraatioiden testaus on suuresti riippuvainen integraatiostrategian lähestymistavasta. Tällaisia lähestymistapoja ovat esimerkiksi bottom-up, top-down sekä big-bang. (ISTQB, 2023)

Järjestelmätestauksessa keskitytään kokonaisen järjestelmän tai tuotteen kokonaiskäyttäytymiseen sekä -ominaisuuksiin. Järjestelmätestaukseen sisältyy usein sekä toiminnallisia end-to-end-testauksia, että ei-toiminnallisia

laatuominaisuuksien testauksia. Joidenkin ei-toiminnallisten laatuominaisuuksien, kuten esimerkiksi sovelluksen käytettävyyden osalta, on parempi toteuttaa testaus koko järjestelmällä käyttöä edustavassa testiympäristössä. Myös osajärjestelmien simulointi on mahdollista. Järjestelmätestauksen voi suorittaa järjestelmän kehitysryhmästä riippumaton testiryhmä. (ISTQB, 2023) Huizinga ja Kolawa (2007) korostavat, että järjestelmätestauksessa sekä toiminnalliset että ei-toiminnalliset ominaisuudet on testattava. Joissakin tilanteissa testattavia järjestelmän laatuominaisuuksia voi olla useita. Ominaisuuksia voivat olla järjestelmän konfiguraatio, suorituskyky, turvallisuus, luotettavuus, palautuminen sekä käytettävyys.

Järjestelmän integraatiotestauksessa keskitytään testattavan järjestelmän, muiden eri järjestelmien sekä ulkoisten palvelujen välisten rajapintojen testaamiseen. Järjestelmän integraatiotestaus edellyttää sellaisia testaukselle sopivia testausympäristöjä, jotka vastaavat mahdollisimman paljon järjestelmän oikeaa käyttöympäristöä. (ISTQB, 2023) Huizinga ja Kolawa (2007) määrittävät järjestelmän integraatiotestauksen tehtäväksi järjestelmään liittyvien elementtien oikean toiminnan todentamisen. Tämä tapahtuu testauksen järjestelmällisellä etenemisellä, jossa kaikki erikseen testatut järjestelmän osat yhdistetään ja testataan yhdessä, kunnes koko järjestelmä on integroitu.

Hyväksymistestauksessa keskitytään järjestelmän validointiin sekä käyttöönottovalmiuden osoittamiseen. Tällä tarkoitetaan, että testattava järjestelmä täyttää käyttäjän sille asettamat liiketoiminnalliset tavoitteet sekä tarpeet. Ihannetilanteessa hyväksymistestauksen suorittavat sovelluksen tulevat käyttäjät. Testauksen päämuotoja ovat käyttäjän hyväksymistestaus (User Acceptance Testing, UAT), toiminnallinen hyväksymistestaus, sopimusperusteinen ja lainasäädännöllinen testaus, sekä alfa- ja beta-testaus. (ISTQB, 2023)

Hyväksymistestauksen tavoitteena on osoittaa, että järjestelmä täyttää kaikki sille asetetut vaatimukset. Testausvaihe on yleinen sopimustilanteissa, joissa hyväksymistestien onnistunut suorittaminen velvoittaa ostajan hyväksymään järjestelmän. Sisäisessä it-kehitystyössä hyväksymistestien onnistunut suorittaminen käynnistää ohjelmiston käyttöönoton tuotantoympäristössä. Testaus voi hyödyntää ajantasaista dataa, ympäristöjä sekä käyttäjäskenaarioita, ja testaus keskittyy järjestelmälle tyypillisimpiin käyttöskenaarioihin, eikä äärimmäisiin olosuhteisiin. (Black, 2009)

2.2 Testaustyypit

Tässä luvussa käydään läpi erilaisia ohjelmistotestauksen tyyppejä, ja avataan sitä, mihin asioihin eri testaustyypit keskittyvät, ja miten ne eroavat toisistaan. Esiteltäviä testaustyyppejä ovat funktionaaliset ja ei-funktionaaliset testit, sekä musta laatikko- ja valkoinen laatikko-testaus.

Funktionaalisessa, eli toiminnallisessa testauksessa, arvioidaan ne toiminnot, jotka komponentin tai järjestelmän pitäisi suorittaa. Toiminnoilla

tarkoitetaan toisin sanoen sitä, mitä testattavan kohteen pitäisi tehdä. Funktionaalisen testauksen päätavoitteena on funktionaalisen täydellisyyden, oikeellisuuden sekä tarkoituksenmukaisuuden tarkistamisen. (ISTQB, 2023) Everettin ja McLeodin (2007) mukaan toiminnallisen testauksen tavoitteet kiteytyvät juuri järjestelmän käyttäytymisen validointiin järjestelmävaatimuksissa dokumentoituja liiketoiminnallisia vaatimuksia vasten. Järjestelmän toiminnallinen testaus saavutetaan sarjalla testejä, jotka käsittelevät yhä suurempaa osaa järjestelmän niistä toiminnoista, jotka mahdollistavat suoraan järjestelmän käyttäjien päivittäisten liiketoimintarutiinien suorittamisen.

Ei-funktionaalinen testaus ei sisällä järjestelmän ydintoimintojen testausta, vaan testauksessa tarkastellaan sitä, miten järjestelmä käyttäytyy suorituskyvyn, luotettavuuden, käytettävyyden sekä turvallisuuden näkökulmasta (Chevutury ym., 2022). Ei-funktionaalisessa testauksessa arvioidaan yksittäisen komponentin tai järjestelmän muita kuin funktionaalisia, eli toiminnallisia, ominaisuuksia. Ei-funktionaalisen testauksen tarkoituksena on selvittää, miten hyvin järjestelmä käyttäytyy. Testauksen päätavoitteena on tarkistaa järjestelmän tai ohjelmiston ei-funktionaaliset laatuominaisuudet. (ISTQB, 2023) ISO/IEC 25010-standardissa (n.d.) ei-funktionaaliset ohjelmiston laatuominaisuudet luokitellaan seuraavasti:

1. Suorituskyvyn tehokkuus
2. Yhteensopivuus
3. Käytettävyys
4. Luotettavuus
5. Turvallisuus
6. Ylläpidettävyys
7. Siirrettävyys

Joissain tilanteissa on tarkoituksenmukaista, että ei-funktionaalinen testaus aloitetaan jo järjestelmän alkuvaiheessa, esimerkiksi osana tarkistuksia ja komponenttien testausta tai järjestelmätestausta. Monet ei-funktionaaliset testit on johdettu funktionaalisista testeistä, koska niissä käytetään samoja toiminnallisia testejä. Kuitenkin sillä erotuksella, että tarkistuksessa keskitytään havainnoimaan, täyttyykö jokin ei-funktionaalinen rajoitus. Tällaisia tarkistuksia ovat esimerkiksi se, suoriutuuko toiminto tietyssä ajassa, tai vastaavasti onko toiminto siirrettävissä uudelle alustalle. Ei-funktionaalisten vikojen myöhäinen havaitseminen voi olla vakava uhka projektin onnistumiselle. (ISTQB, 28-29. 2023)

Musta laatikko-testaus (Black-box testing), tai käyttäytymistestaus, perustuu testauskohteen määrittelyyn ja johtaa testit testauskohteen ulkopuolisesta dokumentaatiosta. Testaustavan päätavoitteena on järjestelmän käyttäytymisen testaaminen sen määrittelyjen pohjalta. Käyttäytymistä analysoidaan ilman viittauksia testattavan kohteen sisäiseen rakenteeseen. (ISTQB, 29 & 38. 2023) Testaajat käyttävät ohjelmiston käyttäytymistestejä löytääkseen virheitä korkean tason toiminnoista, kuten tärkeimmistä ominaisuuksista, toimintaprofiileista sekä todennäköisistä asiakasskenaarioista.

Käyttäytymistestaus edellyttää testaajaltaan yksityiskohtaista ymmärrystä sovelluksen soveltamisalueesta, liiketoimintaongelmasta, jonka sovelluksen on tarkoitus ratkaista, sekä tehtävästä, jota järjestelmä palvelee. (Black, 2009)

Musta laatikko-testauksen ensisijainen tehtävä on tunnistaa koodin toiminnallisuuteen liittyvät ongelmat. Koska tämäntyyppisessä testauksessa jokainen testattava yksikkö tarkistetaan sekä erikseen että osana järjestelmää, voivat kehittäjät testauksen avulla määrittää uudelleen jokaisen sellaisen ohjelmiston osan toiminnallisuuden, joka ei vastaa vaatimuksia. Testaus mahdollistaa myös sellaisten mahdollisten vikojen tunnistamisen, jotka voivat vaikuttaa kielteisesti järjestelmän käyttäytymiseen. (Huizinga & Kolawa, 256. 2007)

Valkoinen laatikko-testaus (White box-testing), tai rakennetestaus, perustuu vastavuoroisesti testattavan järjestelmän rakenteeseen ja johtaa toteutettavat testit järjestelmän toteutuksesta tai sisäisestä rakenteesta, kuten esimerkiksi koodista, arkkitehtuurista, työnkulusta tai tietovirroista. Testaustavan päätavoitteena on testeillä kattaa taustalla oleva rakenne hyväksyttävällä tasolla. (ISTQB, 29. 2023) Rakennetestaus edellyttää yksityiskohtaista teknistä tietämystä testattavasta järjestelmästä. Testaajat laativat rakennetestejä tarkastelemalla itse koodia sekä sen käyttämiä tietorakenteita. Rakennetestien toteutus sopiikin parhaiten testattavia rakenteita toteuttavien kehittäjien tehtäväksi, sillä he ovat parhaiten perehtyneitä testattaviin ominaisuuksiin. (Black, 2009)

White box-testaus käsittää joukon ”mitä jos?” -kysymyksiä, joilla pyritään määrittelemään, käyttäytyykö sovellus edelleen asianmukaisesti epätavallisissa tai poikkeuksellisissa olosuhteissa. Testauksen onnistuminen riippuu kehittäjän kyvystä luoda joukko testitapauksia, jotka ovat riittävän laajoja ja monipuolisia, jotta ne kattavat testattavan yksikön eri haarat ja polut mahdollisimman täydellisesti. Näin on mahdollista paljastaa odottamattomia käytösmalleja. Tällaisten testien luominen on kutienkin haasteellista, sillä mahdollisten testitapausten määrä voi olla huomattavan suuri, ja kaikkien tapausten testaaminen on usein sekä epäkäytännöllistä että käytännössä mahdotonta. (Huizinga & Kolawa, 252-253. 2007)

2.3 Testausautomaatio

Testausautomaatiossa hyödynnetään erilaisten testiskriptien ajamista, joiden avulla voidaan automatisoida manuaalisia testauksen työvaiheita. Testiskriptit pohjautuvat yleensä manuaalisen testauksen vaiheessa esiin nousseisiin testitapauksiin. (Amannejad ym, 2014) Automaatiotestauksen tavoitteena on ohjelmistokehityksen tehokkuuden sekä luotettavuuden kasvattaminen automatisoimalla tehtäviä, jotka manuaalisesti tehtynä olisivat arkipäiväisiä, aikaa vieviä sekä virhealttiita. Tämä tarkoittaa esimerkiksi koodausstandardien noudattamista, testien suorittamista sekä raporttien tuottamista näiden automatisoitujen prosessien tulosten perusteella. Automatisoimalla toistuvia

tehtäviä sekä projektiin liittyvien tietojen keräämistä tarjotaan puitteet tietoon perustuvien päätösten tekemiselle, mikä strukturoidumpaa ja tietoon perustuvaa projektihallintaa. (Huizinga & Kolawa, 2007)

Automatisoitu suorituskykytestaus on olennaisen tärkeää jatkuvassa kehitystyössä sekä jatkuvan integroinnin ympäristössä, joissa nopea palaute sekä kyky tunnistaa ja korjata suorituskykyongelmat nopeasti ovat ratkaisevia kehitystahdin ylläpitämiseksi ja ohjelmiston laadun varmistamiseksi. (Luo ym., 2017)

Automaatiotestauksesta saavutettavat hyödyt verrattuna manuaaliseen testaukseen rinnastetaan usein automaattisesti ajallisiin sekä rahallisiin säästöihin. Fewsterin ja Grahamin (1994) mukaan testien automatisoinnista on kuitenkin saavutettavissa hyötyjä vain silloin, kun toteutettavat testit sekä valitaan että toteutetaan huolellisesti. Testien automatisointi vaikuttaa vain siihen, kuinka taloudellisia sekä kehityskelpoisia testit ovat. Kun testit on kerran toteutettu, on niiden ajaminen yleensä paljon taloudellisempaa, sillä sen suorittaminen maksaa vain murto-osan vastaavasta manuaalisesta työstä. Automaatiotestit ovat kuitenkin yleensä kalliimpia luoda sekä ylläpitää. Tällöin, mitä paremmin testien automatisointia lähestytään, sitä halvemmaksi niiden toteuttaminen tulee pitkällä aikavälillä. Jos ylläpitoa ei oteta huomioon testien automatisoinnissa, voi koko automaatiotestien sarjan päivittäminen maksaa yhtä paljon tai jopa enemmän kuin kaikkien testien suorittaminen manuaalisesti.

Amannejad ym. (2014) listaavat tutkimuksessaan näkökulmia siihen, mitä asioita kannattaa ottaa huomioon, kun harkitaan automaatiotestien toteuttamista. Ensimmäisenä he mainitsevat testattavan kohteen muutosnopeuden. Mitä enemmän testattavaan kohteeseen tulee tulevaisuudessa kohdistumaan muutoksia, sitä enemmän testausautomaation ylläpito tulee kustantamaan. Toinen huomioitava asia on testauksen suoritustiheys: kuinka usein testausta halutaan suorittaa, ja mikä on suhde yksittäisen testituloksen merkityksen sekä tuloksesta maksettavan hinnan välillä? Kolmas huomio on automaation hyödyllisyys. Jos automatisoitavilla testeillä ei ole jatkuvaa arvoa esimerkiksi vikojen löytämiseksi tai ohjelmiston erilaisten skenaarioiden todentamiseksi, onko silloin kannattavaa toteuttaa testausautomaatiota. Näiden lisäksi erilaiset testaustyypit soveltuvat automatisointiin paremmin kuin toiset. Esimerkiksi erilaiset rasitus-, luotettavuus- sekä regressiotestaukset soveltuvat hyvin automatisoitaviksi testeiksi.

3 SUORITUSKYKYTESTAUS

Tässä kappaleessa käsitellään suorituskykytestausta, mikä voidaan nähdä yhtenä ohjelmistotestauksen osa-alueena. Kappaleessa määritellään suorituskykytestauksen eri tavoitteita, sekä käydään läpi suorituskykytestauksen eri tyyppejä sekä näiden tavoitteita. Kappaleessa käsitellään myös suorituskyvyn pullokauloja, joita suorituskykytestauksella pyritään havaitsemaan. Tämän lisäksi esitellään tapoja pullonkaulojen tunnistamiseen sekä niiden suorituskykyvaikutusten lieventämiseen.

Suorituskykytestauksella tarkoitetaan prosessia, jonka avulla pyritään selvittämään, suoriutuuko tietty tuote hyvin ja odotetun laisesti hyvin erilaisissa työmäärissä. Suorituskykytestauksen toteuttaminen on hyvin tuotekohtaista, ja sen sisältö sekä tavoitteet voivat vaihdella eri tuotteiden ja kontekstein välillä. Pääasiassa suorituskykytestauksen tavoitteena on havaita parametreja, jotka aiheuttavat tuotteeseen jonkinlaisia toimintahäiriöitä. Näitä parametreja pyritään tunnistamaan mahdollisimman aikaisessa vaiheessa tuotteen elinkaarta suurempien ongelmien välttämiseksi tulevaisuudessa. (*Mitä on Suorituskykytestaaminen? Tyypit, Käytännöt, Työkalut Ja Paljon Muuta!*, n.d.)

Suorituskykytestit suunnitellaan sen varmistamiseksi, että järjestelmä kykenee käsittelemään sille kohdennettua ja tarkoitettua kuormitusta. Tällä edellytetään sellaisten testisarjojen suorittamista, joissa kuormitusta kasvatetaan pisteeseen, jossa testattavan järjestelmän suorituskykyä ei voida enää hyväksyä. Kuten muissakin testaustyypeissä, on suorituskykytesteissäkin tarkoituksena lopulta löytää järjestelmän puutteet ja ongelmat, sekä varmistaa järjestelmän kyky suoriutua sille asetetuista vaatimuksista. Nämä mahdolliset puutteet löydetään parhaiten asettamalla suorituskykytesteille sellaisia vaatimuksia, jotka ovat suurempia kuin järjestelmälle suunnitellut suorituskykyrajat. (Sommerville, 2011)

3.1 Suorituskykytestauksen tavoitteet

Suorituskykytestauksen tavoitteena on saavuttaa sekä ylläpitää järjestelmälle asetettu haluttu laatutaso, sekä varmistaa kyseisen järjestelmän käytettävyys

potentiaalisten käyttäjien keskuudessa. Suorituskykytestaukselle on asetettu myös erityistarkoituksia, joilla arvioidaan tarkemmin järjestelmän käytettävyyttä, suorituskykyä, sekä kykyä täyttää kansainvälisiä standardeja. Järjestelmän toiminnallisen soveltuvuuden sekä käytettävyyden arvioinnissa on tavoitteena arvioida järjestelmän käytettävyysominaisuuksia voimakkaan kuormituksen ja pitkäaikaisen käytön olosuhteissa. Tavoitteena on tunnistaa, täyttääkö testattava järjestelmä sille asetetut toiminnalliset vaatimukset sekä käytettävyysodotukset järjestelmän todellisissa käyttöympäristöissä. (Melkozerova & Rassomakhin, 2020)

Testattavan järjestelmän suorituskykyä arvioidaan myös kansainvälisen ISO 25010 standardin mukaisesti. Tämän tavoitteena on varmistaa se, että järjestelmä täyttää maailmanlaajuisesti tunnustetut laatua ja suorituskykyä koskevat vertailuarvot. Järjestelmän suorituskykyä kuvaavia pääindikaattoreita on artikkelissa listattuna kolme kappaletta: vasteajan arviointi, resurssien käyttö sekä läpäisykyky. (The International Organization for Standardization (ISO) & The International Electrotechnical Comisison (IEC), n.d.)

Vasteajan arviointi on yleisin suorituskykytestauksessa käytettävä mittari. Tällä testauksen osa-alueella määritetään eri komponenttien sekä järjestelmän kyky vastata käyttäjälle tai toiselle järjestelmälle tietyssä ajassa, sekä tietyissä olosuhteissa. Järjestelmän vasteaika on kriittisin mittari järjestelmän käyttäjäkokemuksen kannalta, sillä pitkät vasteajat voivat vaikuttaa järjestelmän käytettävyyteen. Resurssien käytössä vastaavasti mitataan käytössä olevien resurssien, kuten esimerkiksi RAM-muistin, saatavuutta tiettyjen suorituskykytestien aikana. Resurssien käytön optimoinnilla on tavoitteena saavuttaa järjestelmän optimaalinen suorituskyky ilman, että järjestelmä tuhlaa resurssejaan tai ylikuormittaa niitä. Järjestelmän läpäisykyky mittaa, kuinka monta tehtävää tai tapahtumaa järjestelmä kykenee käsittelemään tietyssä ajassa. Järjestelmän läpäisykyvyn indikaattorit kertovat, jos järjestelmän käyttäytyminen vaadituilla datamäärillä, kuten esimerkiksi käyttäjien määrillä tai erilaisilla datamäärillä, rajoittavat järjestelmän toimintaa. Tällöin suorituskykytestit voivat olla tarpeen arvioimaan järjestelmän kokonaisarkkitehtuurin toimintaa sekä soveltuvuutta. Järjestelmän läpäisykyky on erityisen tärkeä mittari sellaisissa järjestelmissä, joissa käsitellään suuria määriä tietoja tai pyyntöjä, kuten esimerkiksi tietokantapalveluissa tai verkkosovelluksissa. (Melkozerova & Rassomakhin, 2020; The International Organization for Standardization (ISO) & The International Electrotechnical Comisison (IEC), n.d.)

3.2 Suorituskykytestauksen eri tyypit

Melkozerova ja Rassomakhin (2020) ovat artikkelissaan avanneet suorituskykytestauksen eri tyyppejä. Testauksen eri tyyppejä on useita, ja kullakin testityypillä on omat tavoitteensa. Taulukossa 1. on listattuna

artikkelissa esiteltyjä järjestelmän suorituskykyä mittaavia eri suorituskykytestauksen testityyppejä.

Kuormitustestauksessa (Load Testing) järjestelmän suorituskykyä mitataan asteittain kasvavan kuormituksen alla. Tämä tarkoittaa, että testissä simuloidaan järjestelmän käyttäjämäärien tai pyyntöjen lisääntymistä ja tutkitaan, miten hyvin järjestelmä pystyy käsittelemään kyseisen kuormituksen. Testauksen tavoitteena on ensisijaisesti varmistaa, että järjestelmä pysty toimimaan odotetun kuormituksen alla. Toisena tavoitteena on löytää mahdollisia rajoja, joiden ylityksestä järjestelmän suorituskyky heikkenee.

Stressitestauksessa (Stress Testing) keskitytään järjestelmän suorituskyvyn mittaamiseen äärimmäisissä olosuhteissa, kuten erittäin suurien käyttäjämäärien tai pyyntöjen alla. Testauksessa ylitetään usein järjestelmän odotetut käyttörajat, minkä tarkoituksena on havainnoida sitä, miten testattava järjestelmä reagoi ylikuormitukseen. Stressitestauksen avulla on mahdollista havaita järjestelmässä piileviä pettämispisteitä ja varmistaa, että järjestelmä pystyy käsittelemään odottamattomia kuormitushuippuja ilman, että se menettää merkittävästi suorituskykyään tai kaatuu kokonaan.

Skaalautuvuustestauksessa (Scalability Testing) tarkastellaan järjestelmän kykyä kasvattaa suorituskykyään käsiteltävien resurssien, kuten käyttäjämäärien tai tietomäärien, kasvaessa. Testin tavoitteena on selvittää, onko järjestelmällä kykyjä selviytyä sekä tehokkaasti että ilman suorituskykyongelmia laajennetuissa olosuhteissa. Skaalautuvuustestauksen avulla voidaan arvioida, kuinka hyvin testattava järjestelmä voi vastata kasvaviin vaatimuksiin sekä mukautua suurempaan käyttöön.

Volyymitestauksessa (Volume Testing) tarkastellaan järjestelmän kykyä käsitellä suuria tietomääriä. Tämän testaustyypin hyödyntäminen on tärkeää erityisesti erilaisissa tietokantajärjestelmissä tai järjestelmissä, joissa käsitellään erittäin suuria datamääriä. Tavoitteena on varmistaa, että järjestelmä kykenee suurien datamäärien käsittelyyn ilman merkittävää suorituskyvyn heikkenemistä tai järjestelmän kaatumista.

Piikkitestaus (Spike Testing) keskittyy testaustyyppinä testaamaan järjestelmän suorituskykyä tilanteissa, joissa järjestelmään kohdistuu äkillisiä kuormitushuippuja. Testin tarkoituksena on testata järjestelmän reagointikykyä äkillisiin kuormitustilanteisiin, sekä tarkastella järjestelmän kykyä palautua sen normaaliin toimintatilaan kuormituksen vähentyessä.

Kestävyystestauksessa (Endurance Testing) järjestelmää testataan pitkään jatkuvalla korkealla kuormituksella. Tämän avulla voidaan havaita mahdolliset resurssiongelmat, kuten muisti- tai yhteysongelmat, jotka voivat ajan myötä aiheuttaa ongelmia järjestelmän suorituskykyyn. Hyödyntämällä tätä testaustyyppiä voidaan varmistua siitä, että järjestelmä säilyttää suorituskykynsä sekä toimintavakautensa myös pitkäaikaisessa käytössä ilman, että sen resurssit loppuvat.

Luotettavuustestauksessa (Reliability Testing) on tavoitteena varmistaa, että järjestelmä kykenee suorittamaan toimintojaan oikein määritellyissä

olosuhteissa joko tietyn ajan tai määrätyn määrän eri operaatioita. Testityypillä arvioidaan järjestelmän vakautta ja toimintavarmuutta pidemmällä aikavälillä.

Rinnakkaistestauksessa (Concurrency Testing) keskitytään arvioimaan järjestelmän suorituskykyä tilanteissa, joissa. Suuria määriä käyttäjiä suorittaa toimintoja samanaikaisesti. Testauksella simuloidaan tilannetta, jossa syntyy kilpailua järjestelmän käytettävissä olevista resursseista, kuten muistista tai tietokantayhteyksistä. Testin tehtävänä on arvioida, kuinka hyvin järjestelmä selviää tällaisista kilpailutilanteista. Testissä voidaan myös tarkastella monisäikeisten sovellusten synkronointia ja varmistaa niiden toimivuus sekä oikeellisuus, vaikka useita operaatioita suoritetaankin samanaikaisesti.

Kapasiteettitestauksen (Capacity Testing) tavoitteena on määrittää, kuinka paljon käyttäjiä tai prosesseja järjestelmä voi palvella, ennen kuin järjestelmän suorituskyky alkaa heikentyä. Testin avulla voidaan tunnistaa järjestelmän eri kapasiteettien rajat, ja tätä kautta arvioida sitä, milloin järjestelmä tulisi tarvitsemaan laajennuksia tai päivityksiä.

Taulukko 2. Suorituskykytestauksen eri tyypit

Testaustyyppi	**Selite**
Kuormitustestaus (Load)	Arvioidaan järjestelmän kykyä käsitellä kasvavaa odotettua kuormitusta ja tutkitaan sen kestävyyttä.
Stressitestaus (Stress)	Määritellään järjestelmän kyky ylläpitää huippukuormitusta yli odotettujen rajojen ja tutkitaan sen käyttäytymistä epänormaaleissa kuormitusmuutoksissa.
Skaalatutuvuuden testaus (Scalability)	Tarkistetaan järjestelmän kyky parantaa suorituskykyään suhteessa siihen, että käytettävissä on enemmän resursseja.
Volyymitestaus (Volume)	Tutkitaan järjestelmän suorituskykyä erilaisten tietomäärien käsittelyssä.
Piikkitestaus (Spike)	Arvioidaan järjestelmän reagointia äkillisiin kuormituspiikkeihin ja sen palautumista normaalitilaan.
Kestävyystestaus (Endurance)	Keskitytään järjestelmän vakauteen tunnistamalla suorituskykyyn vaikuttavia tai vikoja aiheuttavia resursseihin liittyviä ongelmia.
Luotettavuustestaus (Reliability)	Samanlainen kuin kestävyystestaus, mutta siinä tutkitaan sovelluksen kykyä suorittaa toimintojaan tietyissä olosuhteissa tietyn ajan kuluessa.
Rinnakkaistestaus (Concurrency)	Tarkastellaan järjestelmän käyttäytymistä tietyissä samanaikaisissa toiminnoissa ja tutkitaan samanaikaisten pyyntöjen vaikutusta järjestelmän resursseihin.
Kapasiteettitestaus (Capacity)	Arvioidaan järjestelmän kapasiteettia käsitellä operaatioita ja pyritään tunnistamaan mahdolliset kehitys- ja laajennuskohteet.

3.3 Suorituskyvyn pullonkaulat ja niiden tunnistaminen

Tässä alaluvussa käsitellään suorituskyvyn pullonkauloja sekä mahdollisia keinoja niiden tunnistamiseksi. Tämän lisäksi luvussa käydään läpi esimerkkejä keinoista, joilla tunnistettujen pullonkaulojen vaikutusta järjestelmän suorituskykyyn voidaan lieventää.

Luo ym. (2017) käsittelevät artikkelissaan järjestelmien pullonkauloja, joita suorituskykytesteillä pyritään havainnoimaan. Suorituskyvyn pullonkaulat ovat järjestelmän kohtia, joissa sen suorituskyky heikkenee merkittävästi. Tämä johtuu usein odottamattomista tai intensiivisistä laskentavaatimuksista tietyille

syöttöarvon yhdistelmille. Näiden pullonkaulojen tunnistaminen on ratkaisevan tärkeää järjestelmän suorituskyvyn optimoimiseksi, seillä ne voivat johtaa ominaisuuksien heikkenemiseen. Heikentyminen voi ilmetä esimerkiksi järjestelmän pidempänä vasteaikana tai pienentyneenä läpäisykykynä tietyissä työkuormituksissa. Artikkelissaan kirjoittajat korostavat pullonkaulojen tunnistamisen haastetta automaatiotestien avulla, mikä on yksi suorituskykytestauksen keskeisimpiä tavoitteita. Perinteinen lähestymistapa näiden pullonkaulojen löytämiseen edellyttää laajaa manuaalista testausta monilla toimintojen sekä syöttötietojen yhdistelmillä, mikä on sekä aikaa vievää että tehotonta etenkin monimutkaisissa järjestelmissä.

Siddiqui ja Woodside (2002) kuvaavat artikkelissaan pullonkaulojen syntymistä seuraavasti. Pullonkaula syntyy tilanteissa, joissa yksi järjestelmän komponentti hidastaa koko järjestelmän toimintaa, koska se ei kykene käsittelemään sille asetettua kuormaa tehokkaasti. Kirjoittajat esittelevät myös järjestelmän herkkyysanalyysin menetelmänä pullonkaulojen tunnistamiseen. Herkkyysanalyysin tarkoituksena on tutkia, kuinka pienet muutokset järjestelmän eri komponenteissa vaikuttavat järjestelmän kokonaissuorituskykyyn. Analyysin avulla voidaan näin tunnistaa ne komponentit, jotka ovat järjestelmän suorituskyvynkannalta kaikkein kriittisimmät. Tilanteessa, jossa pieni muutos resurssikysynnässä aiheuttaa merkittävän vaikutuksen järjestelmän vasteaikaan tai läpäisykykyyn, saattaa testattava komponentti muodostaa todennäköisesti pullonkaulan.

Kun järjestelmän mahdollisia pullonkauloja on saatu tunnistettua, on seuraavana tehtävänä näiden lieventäminen tai poistaminen kokonaan. Neilson ym. (1995) listaavat artikkelissaan strategioita, joilla pullonkauloja voidaan lieventää. Artikkelissa pullonkauloja sekä suorituskykyä käsitellään asiakas-palvelinarkkitehtuurin näkökulmasta, ja eri strategioita ovat artikkelin mukaan järjestelmän monisäikeistäminen ja kloonien käyttö, resurssien lisääminen sekä ohjelmistorakenteen optimointi. Lisäämällä klooneja eli kopiota samasta tehtävästä, voidaan järjestelmän suorituskykyä parantaa, kun vastaava kuormitus jakautuu nyt useampaan eri instanssiin. Tällöin alkuperäisen tehtävän pullonkaula voi lieventyä, ja järjestelmän läpäisykyky parantua. Kloonien lisäämistä voidaan hyödyntää erityisesti tilanteissa, joissa esimerkiksi palvelin on ohjelmistollisesti ylikuormitettuna, mutta sen alla olevat resurssit, kuten esimerkiksi prosessori- tai muistikapasiteetti, eivät ole vielä täydessä käytössä. Toinen strategia pullonkaulojen lieventämiseen on lisätä käytössä olevia resursseja. Tämä strategia toimii kuitenkin ainoastaan sellaisissa tilanteissa, joissa pullonkaulan juurisyy liittyy fyysisten resurssien, kuten prosessorien tai muistikapasiteetin, rajallisuuteen. Kolmantena strategiana kirjoittajat listaavaan järjestelmän ohjelmistorakenteen optimoinnin. Ohjelmistorakenteessa voi esimerkiksi ilmetä kriittisiä osia, jotka vaativat synkronointia tai yksinoikeutta resursseihin, ja tätä kautta hidastaa järjestelmän suorituskykyä. Optimoimalla ohjelmistorakennetta esimerkiksi parantamalla synkronointimekanismeja tai uudelleenjärjestelemällä eri tehtäviä, voidaan lieventää pullonkaulojen vaikutusta järjestelmän suorituskykyyn.

4 Suorituskyvyn budjetointi

Tässä kappaleessa määritellään suorituskykybudjettia sekä sen mahdollisia mittareita. Käsitteitä pyritään avaamaan aiheeseen tehtävän kuvailevan kirjallisuuskatsauksen muodossa. Katsauksen pohjalta muodostetaan käsite siitä, mikä suorituskykybudjetilla voidaan tarkoittaa sovelluskehityksen kontekstissa, ja mitä suorituskyvyn budjetoinnista on tutkimuksen kontekstissa kirjoitettu. Aiheen tueksi kappaleessa käsitellään budjettia sekä budjetointia yleisesti.

Kuvailevan kirjallisuuskatsauksen prosessi voidaan jakaa Kangasniemen ym. (2013) mukaan neljään eri vaiheeseen: tutkimuskysymyksen muodostamiseen, aineiston valintaan, kuvailun rakentamiseen sekä tulosten tarkasteluun. Prosessin ensimmäinen osa on toteutettu tutkielman alussa, jolloin asetettiin tutkielman tutkimuskysymykset. Kirjallisuuskatsauksen avulla pyritään vastaamaan tutkimuskysymykseen suorituskyvyn budjetoinnista. Tutkimuskysymyksen pohjalta on toteutettu aineiston valinta. Aineiston valinnassa pyrittiin löytämään artikkeleita sekä julkaisuja, jotka käsittelevät suorituskykybudjettia ohjelmistokehityksen kontekstissa tai suorituskyvyn budjetointia yleisesti. Koska aineiston keruuvaiheessa tunnistettiin, että aiheesta on varsin vähän julkaisuja, lisättiin aineistoon budjettia sekä budjetointia yleisesti käsittelevää kirjallisuutta. Varsinaisen kuvailun rakentaminen sekä tulosten tarkastelu on toteutettu tämän luvun seuraavissa alaluvuissa, joissa käsitellään kerätyn aineiston pohjalta budjetointia yleisesti, suorituskykybudjettiin sekä sen laadintaan liittyviä käsitteitä sekä esimerkkejä mahdollisista suorituskykybudjetin mittareista.

4.1 Budjetti ja budjetointi käsitteinä

Budjetti sekä budjetointi ovat yleisiä talouteen sekä liiketoimintaan liittyviä termejä. Tutkielman kontekstissa käsiteltävän ohjelmistojen suorituskykytestaukseen liittyvän suorituskykybudjetin tueksi tässä alaluvussa käydään läpi sitä, mitä budjetista sekä budjetoinnista on talouden sekä liiketoiminnan alan teoksissa kirjoitettu. Kirjassaan *Laskentatoimi* (2015) Jormakka ym. määrittelevät budjetin sekä budjetoinnin seuraavalla tavalla.

Budjetti on numeroin ilmaistu toimintasuunnitelma tietylle yrityksen ajanjaksolle, yleensä tulevalle vuodelle tai tilikaudelle. Budjetin laadinta pohjautuu yrityksen strategiaan sekä pitkän aikavälin suunnitelmiin, ja laadinnassa huomioidaan yrityksellä käytössään olevat resurssit, aikataulut sekä työtehtävien jakaminen. Budjetoinnin kirjoittajat määrittelevät tehtäviksi, jotka käsittävät budjetin laadintaa, budjettien ohjattua käyttöä, erojen analysointia sekä korjaavien toimenpiteiden suunnittelua ja toteuttamista. Budjetoinnin tavoitteena on pyrkiä tavoitteellisen toiminnan suunnitteluun sekä valvontaan, sekä samalla koordinoida yrityksen eri osa-alueita. Tällä tavoin pyritään yrityksen eri osastojen väliseen yhteistyöhön yhteisten tavoitteiden sekä näkemysten saavuttamiseksi.

4.2 Suorituskykybudjetin määritelmä

Suorituskykybudjetista on käsitteenä varsin vähän tieteellisiä julkaisuja. Monet aihetta käsittelevät lähteet ovat erilaisia artikkeleita, blogikirjoituksia tai suorituskykyyn liittyvien yritysten kirjoittamia artikkeleita suorituskykybudjetin merkityksestä sekä sen hyödyntämisestä ohjelmistojen ja verkkosivujen tehokkuuden valvonnassa sekä parantamisesta. Tässä kappaleessa on koottu edellä mainittujen lähteiden kaltaisista materiaaleista katsaus siitä, miten niissä suorituskykybudjettia sekä sen käyttöä on määritelty.

MDM Wed Docs-sivusto (2023) kuvailee dokumentaatiossaan suorituskykybudjettia rajaksi, jonka tarkoituksena on estää ohjelmiston regressiota. Tällaiset budjetissa määriteltävät raja-arvot voivat kohdistua esimerkiksi yksittäisiin tiedostoihin, tiettyihin mittareihin, tai johonkin mukautettuun mittariin. Budjetin olemassaolon tarkoituksena on kuvastaa saavutettavia tavoitteita, ja se on kompromissi hyvän käyttäjäkokemuksen sekä ohjelmiston tehokkuuden välillä. Suorituskykybudjetin perimmäinen tarkoitus on korreloida suorituskyvyn vaikutusta liiketoiminnan tai tuotteen tavoitteisiin.

Mihajlija määrittelee artikkelissaan ”Performance budgets 101” (web.dev, 2018) suorituskykybudjetiksi joukkoa määriteltyjä mittareita sekä rajoituksia, jotka on asetettu sivuston suorituskykyyn vaikuttaville mittareille. Tällaisia mittareita voivat olla esimerkiksi sivun kokonaiskoko, latausaika mobiiliverkossa tai lähetettyjen http-kutsujen määrä. Suorituskykybudjetin määrittäminen auttaa nostamaan esiin keskustelua tuotteen verkkosuorituskyvystä, ja se toimii viitekehyksenä suunnittelua, tekniikkaa ja ominaisuuksien lisäämistä koskevien päätösten tekemiselle. Budjetin tärkein ajatus on asettaa tavoitteet, jotta kehittäjät voivat tasata suorituskykyä ilman, että ohjelmiston toiminnallisuus tai käyttökokemus kärsii.

Artikkelissaan ”Performance Aware Software Development (PASD) Using Resource Demand Budgets” Siddiqui ja Woodside (2002) määrittävät ohjelmiston suorituskyvyn budjetin joukoksi ennalta määriteltyjä rajoja eri resurssien käytölle tiettyjen ohjelmiston komponenttien tai sen suorittamien operaatioiden osalta. Budjetin raja-arvojen määrittäminen pohjautuu kirjoittajien

mukaan aikaisempiin kokemuksiin järjestelmästä sekä sille asetettuihin suorituskykyvaatimuksiin. Itse budjetointiprosessissa voidaan soveltaa ns. ”hajoita ja hallitse”-strategiaa (divide and conquer). Tällä tarkoitetaan sitä, että testattava järjestelmä pilkotaan erilaisiin osiin, ja jokaiselle osalle luodaan omat budjettinsa, joita testataan erilaisten suorituskykyä testaavien mallien avulla. Mallien tarkoituksena on varmistaa, että jos asetetut budjettien raja-arvot täyttyvät, täyttää järjestelmä sille asetetut suorituskykytavoitteet. Artikkelissaan kirjoittajat mainitsevat myös hyvän suorituskykybudjetointiprosessin olevan luonteeltaan iteratiivinen. Kun uutta tietoa järjestelmän suorituskyvystä tulee saataville, on tärkeää tarkistaa budjetin sisältöjä sekä hienosäätää raja-arvoja. Näillä keinoin voidaan varmistaa, että järjestelmä pysyy suorituskykytavoitteiden mukaisena koko sen kehityskaaren ajan.

Suorituskykybudjetin laatiminen antaa kehittäjille käyttöön erilaisia vertailuarvoja, joiden avulla kehittäjät pystyvät paremmin seuraamaan ohjelmiston tai verkkosivuston suorituskykyä. Suorituskykybudjetin mittarit mahdollistavat suorituskyvyn analysoinnin tilanteissa, joissa ohjelmiston suorituskyky on laskenut ja laskun aiheuttaneita osa-alueita tulisi kyetä tunnistamaan. (Medium, 2020)

Artikkelissa ”What Is a Performance Budget?” (Crystallize, n.d.) kuvaillaan, että suorituskykybudjetti on pohjimmiltaan joukko rajoja, joista kehitystiimi on yhdessä sopinut. Voidaan tehdä oletus, jossa uuden ominaisuuden tai elementin lisäys aiheuttaisi sen, että verkkosivusto tai ohjelmisto ylittäisi ennalta määritellyn suorituskykybudjetin. Tällöin kehitystiimin on tehtävä muokkauksia joko lisättyyn ominaisuuteen tai muualle ohjelmistoon, jotta ohjelmiston kokonaissuorituskyky pysyy annetuissa rajoissa. Tämä voi tarkoittaa olemassa olevien resurssien optimointia, joidenkin vähemmän kriittisten resurssien tai ominaisuuksien poistamista, tai uuden ominaisuuden tarpeellisuuden harkitsemista uudelleen.

Projektin tarkka suorituskykybudjetti vaihtelee ohjelmiston tai sivuston erityistavoitteiden sekä -tarpeiden mukaan. Esimerkiksi mediapainotteisemmalla sivustolla voi olla suurempi budjetti kuvia varten, kun taas vastaavasti verkkosovelluksessa voi olla suurempi budjetti skriptejä varten. Kaikissa tapauksissa suorituskykybudjetin tarkoituksena on keskittyä ylläpitämään nopeaa, reagoivaa ohjelmistoa tai sivustoa, joka tarjoaa parhaan mahdollisen käyttökokemuksen. Suorituskykybudjetit ovat yhä tärkeämpi työkalu nykyaikaisessa web-kehityksessä, sillä ne auttavat tasapainottamaan toiminnallisuuden ja suorituskyvyn usein kilpailevia etuja.

4.3 Suorituskykybudjetin mittarit

Tässä alaluvussa käydään läpi esimerkkejä mahdollisista suorituskykybudjetissa käytettävistä mittareista. Kuten edellisessä luvussa kävi ilmi, on suorituskykybudjetti sekä sisältöineen että käytänteineen hyvin vapaamuotoinen sekä käyttöympäristöönsä sekä -tarkoituksiinsa muokattu. On

kuitenkin hyvä antaa joitain konkreettisia esimerkkejä kyseisistä mittareista, ja näitä käydäänkin tässä alaluvussa läpi.

Artikkelissa ”What is Performance Budget?” (Crystallize, n.d.) avataan muutamia käytettyjä, web-sovellukselle soveltuvia suorituskyvyn mittareita, joita voidaan hyödyntää suorituskykybudjettia rakennettaessa. Näitä mittareita on avattu taulukossa 2. Suorituskykybudjetti voidaan rakentaa monien eri parametrien mukaan. Erilaisia parametreja voivat olla esimerkiksi resurssikoot, määrään perustuvat mittarit, välitavoitteiden ajoitukset (Milestone Timings), sekä sääntöpohjaiset mittarit (Rule-Based Metrics). Resurssikokoihin liittyvät budjetin parametrit voivat rajoittaa esimerkiksi web-sivuilla käytettyjen resurssien, kuten JavaScriptin, CSS:n, kuvien tai sivun kokonaispainon kokoa. Budjetissa voidaan esimerkiksi määrätä, ettei sivun JavaScript-koodin kokonaismäärä saa ylittää 500 kilotavua. Määrään perustuvien mittarien tarkoituksena on rajoittaa tiettyjen sivun elementtien tai pyyntöjen määrää, kuten esimerkiksi sivun http-kutsujen kokonaismäärää, tai käytettyjen kuvien tai skriptien kokonaismäärää. Välitavoitteiden (Milestones) ajoitukseen liittyvät parametrit perustavat budjetin sivun latausprosessin keskeisiin hetkiin, joita ovat esimerkiksi Time to First Paint (TTFP), Time to Interactive (TTI) sekä First Contentful Paint (FCP). Sääntöpohjaisilla mittareilla (Rule-Based Metrics) rakennetuissa budjeteissa hyödynnetään erilaisia suorituskyvyn tarkastustyökalujen, kuten esimerkiksi Google Lighthouse:n, tuloksia.

Taulukko 3. Esimerkkejä suorituskykybudjetin mittareista web-sovellukselle

Mittari	**Sisältö**
Resurssikokoihin pohjautuvat mittarit	Esimerkiksi Web-sivun resurssikoot: - JavaScript (Skriptit) - CSS (Muotoilut) - Sisältö (kuvat, videot ym.)
Määrä	Esimerkkejä testattavista määristä: - Elementtien määrä sivulla - Sivun esittämien http-pyyntöjen määrä - Scriptien määrä - Sisällön määrä (kuvat, videot)
Välitavoitteet (Milestones)	Latausprosessin eri vaiheet: - Time to First Paint (TTFP) - Time to Interactive (TTI) - First Contentful Paint (FCP)
Sääntöpohjaiset mittarit (Rule-Based Metrics)	Erilaiset suorituskyvyn tarkastustyökalujen hyödyntäminen: - Google Lighthouse - Google PageSpeed Insights - DebugBear

Siddiqui ja Woodside listaavat artikkelissaan (2002) erilaisia suorituskykybudjetissa hyödynnettäviä mittareita, joiden avulla voidaan monitoroida sekä hallita järjestelmän suorituskykyä sekä sen resurssien käyttöä. Kyseisiä mittareita ovat erilaiset aikaviiveet, järjestelmän läpäisykyky ja kapasiteetti sekä saturaatiopisteet.

Aikaviiveet (Time Delays) mittaavat aikaa, mikä kuluu jonkin toiminnon suorittamiseen, kuten esimerkiksi vastauksen saamiseen tai viestin lähettämiseen. Nämä ajat voidaan ilmaista keskiarvoina, keskiarvoina tietyllä varianssilla tai prosenttiperusteisina arvoina, kuten esimerkiksi niin, että 95 % mitatuista vasteajoista tulee olla tietyn aikarajan sisällä. Mahdolliset viiveet vaikuttavat suoraan järjestelmän käyttökokemukseen sekä käytettävyyteen. Läpäisykyvyllä (Throughput) mitataan sitä, kuinka monta toimintoa tai pyyntöä järjestelmä onnistuu käsittelemään tietyssä ajassa. Kapasiteetilla (Capacity) puolestaan viitataan järjestelmän kykyyn käsitellä suuria prosesseja tai käyttäjiä samanaikaisesti ilman, että järjestelmän suorituskyky heikkenee. Tämä mittari on erityisen hyödyllinen sekä tärkeä järjestelmissä, joissa on suuria yhtäaikaisia käyttäjämääriä.

Saturaatiopisteillä (Saturation Points) kuvataan tilanteita, joissa testattavan järjestelmän resurssit saavuttavat maksimaalisen kuormituksen. Kun tämä piste saavutetaan, heikkenee lisäkuorman käsittely merkittävästi ja järjestelmän vasteajat kasvavat.

5 Tutkimusmenetelmä

Tämän tutkimuksen tutkimusongelman ratkaisemisessa sekä siihen liittyvien tutkimuskysymyksiin vastaamisessa hyödynnetään konstruktiivista tutkimusotetta. Konstruktiivisen tutkimusotteen tavoitteena on tuottaa ratkaisu johonkin reaalimaailman ongelmaan innovatiivisen konstruktion kautta. Tutkimuksen kontekstissa reaalimaaliman ongelmana on suorituskykytestauksen toteuttaminen sekä budjetointi tutkimuksen kohdeyrityksen tarpeisiin. Tällaisia konstruktioita voivat olla esimerkiksi erilaiset mallit, rakenteet, suunnitelmat tai muut ihmisen luomat artefaktit. Konstruktioille ominainen piirre on niiden syntytapa: ne eivät ole tutkimuksen avulla löydettyjä, vaan ne keksitään ja kehitetään. Tutkimuksessa kehitettävä konstruktiona voidaan pitää luotavia suorituskykytestejä sekä laadittavaa suorituskykybudjettia, joiden toteutus pohjautuu yhdistelmään aiemmasta tutkimustiedosta sekä tutkijan omasta näkemyksestä itse tutkimusongelmasta. (Lukka, 2014)

5.1 Suunnittelutieteellinen tutkimusmenetelmä

Tutkimuksen tutkimusmenetelmänä hyödynnetään Peffersin ym. (2007) määrittelemää suunnittelutieteellistä tutkimusmenetelmää (eng. design science research). Seuraavaksi käydään läpi kyseisessä tutkimusmenetelmässä toteutettavat vaiheet, niiden sisällöt sekä kuinka vaiheet näkyvät suoritettavassa tutkimuksessa. Peffersin ym. (2007) laatimassa mallissa tutkimusprosessille on määritelty seuraavat vaiheet:

1) Ongelman tunnistaminen ja motivointi
2) Ratkaisun tavoitteiden määrittely
3) Suunnittelu ja kehitys
4) Demonstraatio
5) Arviointi
6) Viestintä

Prosessin ensimmäisessä vaiheessa määritellään tutkimusongelma sekä perustellaan tavoitellun ratkaisun arvo. Toisessa vaiheessa määritellään ratkaisun tavoitteet. Tavoitteiden tulisi olla johdettavissa järkevällä tavalla ongelman määrittelystä. Kolmas vaihe sisältää luotavan artefaktin toiminnallisuuden ja arkkitehtuurin määrittäminen, sekä varsinaisen artefaktin luominen. Neljännessä vaiheessa osoitetaan luodun artefaktin käyttö yhden tai useamman ongelman ratkaisemiseksi. Viidennessä vaiheessa arvioidaan ratkaisun sopivuutta ongelman ratkaisuksi sekä kartoitetaan mahdollisia puutteita, jotka vaatisivat lisää kehitystä. Kuudennen eli viimeisen vaiheen tarkoituksena on viestiä saavutetusta ratkaisusta sekä sen hyödyistä eri kohderyhmille. (Peffers ym., 2007)

Peffersin ym. (2007) mukaan suunnittelutieteellinen menetelmä voidaan aloittaa kohdista 1–4 tutkimuksen tarpeen mukaan. Tässä tutkielmassa tutkimusprosessi käynnistetään kohdasta yksi, eli tutkimuksen lähtökohtana on ongelmakeskeinen lähestymistapa. Menetelmän mukaisesti työssä tunnistetaan olemassa oleva ongelma ja perustellaan tarve sen ratkaisemiseksi. Tutkielman tarkoituksena on ratkaista case-yrityksen tarve varmistua siitä, että case-yrityksen kehittämä ohjelmisto pystyy suoriutumaan korkean kuormituksen tilanteista riittävällä tasolla. Ongelmaksi tunnistettiin siis suorituskyvyn testaus sekä sen hallinta. Motivoivana tekijänä tutkielmassa on parantaa case-yrityksen ohjelmiston luotettavuutta sekä käyttäjäkokemusta.

Tutkielman ongelman pohjalta tutkielmassa esitetään ongelman ratkaisulle asetetut tavoitteet sekä ratkaisun saavuttamiseksi valitut keinot kappaleessa 6. Ongelman ratkaisemiseksi kehitetään tutkimusmenetelmän mukaisesti artefakti, jota tässä tutkielmassa edustavat luodut suorituskykytestit sekä suorituskykybudjetti. Tämän artefaktin tarkoituksena on ratkaista kohdeyrityksen tavoitteet ohjelmiston suorituskyvyn ylläpidosta, testauksesta sekä sen seurannasta, mahdollistaen näin kohdeyritykselle paremmat edellytykset suorituskykyisen ohjelmiston tuottamiseen.

Tavoitteiden ja ratkaisukeinojen pohjautuen tutkimuksessa esitellään seuraavaksi tutkimusmenetelmän mukaisesti tavoitellun artefaktin suunnittelu sekä kehittäminen, joita kuvataan alaluvuissa 6.1 sekä 6.2. Kehittämisen pohjalta saatiin luonnos artefaktista, joka sisältää sekä suorituskykybudjetin että alustavat suunnitelmat suorituskykytestauksen toteuttamiselle. Näillä pyritään vastaamaan tutkimusongelmaan, joka tunnistettiin alaluvussa 6.2.1. Suunnitteluvaiheessa kehitettiin erilaisia suorituskyvyn testausmenetelmiä, joilla voidaan varmistaa ohjelmiston suorituskyvyn valvonta erilaisten kuormitusten alla. Kehitettyjen menetelmien pohjalta tutkielmassa pystyttiin siirtymään tutkimusmenetelmän demonstrointivaiheeseen.

Artefaktin demonstrointi sekä arviointi toteutettiin alaluvussa 6.2.5, jossa kuvattiin senhetkisen artefaktin esittely case-yritykselle. Esittely sisälsi artefaktista suorituskykytestien osuuden, ja esittelyn tavoitteena oli arvioida esiteltävien vaihtoehtojen sopivuutta case-yrityksen tarpeisiin. Demonstraatiovaiheessa arvioitiin esiteltyjen vaihtoehtojen käytännön toimintaa sekä kykyjä valvoa ohjelmiston suorituskykyä sekä valmiuksia tukea

suorituskyvyn ylläpitoa. Arvioinnin pohjalta saatiin rajattua esitellyistä vaihtoehdoista yksi teknologia, jonka pohjalta luotavaa artefaktia päästiin jatkokehittämään. Tämän arvioinnin pohjalta tutkimuksessa siirryttiin iteratiivisesti tutkimusmenetelmän mukaisesti kohtaan 3.

Menetelmän viimeisenä vaiheena viestittiin tutkielman tulokset dokumentoimalla luodut suorituskykytestit sekä suorituskykybudjetti case-yritykselle. Dokumentaatio toteutettiin tämän tutkielman muodossa sekä erikseen luotujen ratkaisujen yhteyteen suoraan case-yritykselle. Dokumentaation tavoitteena oli viestiä kehitettyjen ratkaisujen hyödyistä, mahdollisuuksista sekä kehityskohdista, joita case-yritys voi hyödyntää tuotekehityksessään tulevaisuudessa.

5.2 Kirjallisuuskatsaus

Tässä tutkimuksessa hyödynnetään kirjallisuuskatsausta tutkimusmetodina määrittämään ja kuvaamaan tutkimuksen toista osaa, eli suorituskykybudjettia. Kirjallisuuskatsaus on Salmisen (2011) mukaan tutkimusmetodi, jonka avulla pyritään tekemään 'tutkimuksesta tutkimusta'. Tällä tarkoitetaan jo toteutettujen tutkimusten tulosten koostamista, jotka toimivat perustana jälleen uusille tutkimustuloksille. Kirjallisuuskatsauksen eri perustyyppejä ovat kuvaileva kirjallisuuskatsaus, systemaattinen kirjallisuuskatsaus sekä meta-analyysi. Tutkimuksessa hyödynnetään näistä tutkimusmetodin perustyypeistä kuvailevaa kirjallisuuskatsausta.

Kuvaileva kirjallisuuskatsaus on yleisimmin tutkimuksissa käytetty kirjallisuuskatsauksen perustyyppi, jota voidaan luonnehtia yleiskatsauksesi aiheeseen ilman tiukkoja tai tarkkoja sääntöjä. Katsauksessa käytetyt aineistot ovat laajoja, eivätkä aineiston valintaa rajaa metodit tai säännöt. Tästä huolimatta tutkittava ilmiö pystytään kuitenkin kuvaamaan laaja-alaisesti, ja tutkittavan ilmiön ominaisuuksia pystytään tarvittaessa luokittelemaan. Tutkimuskysymykset ovat väljempiä verrattuna systemaattiseen kirjallisuuskatsaukseen tai meta-analyysiin. Kuvaileva katsaus toimii itsenäisenä tutkimusmetodina, mutta sen voidaan katsoa myös tarjoavan uusia tutkittavia ilmiöitä jälkeenpäin toteutettavaa systemaattista kirjallisuuskatsausta varten. (Salminen, 2011)

6 TULOKSET

Tässä kappaleessa esitellään tutkielman tutkimusmenetelmän avulla saavutetut tulokset suorituskyvyn budjetoinnin sekä suorituskyvyn testauksen menetelmistä. Tutkielmassa noudatettiin Peffersin ym. (2007) määrittelemää suunnittelutieteellistä tutkimusmenetelmää, käsittäen ongelman tunnistamisen, tavoitteiden määrittelyn, artefaktin suunnittelun ja kehittämisen, sekä demonstroinnin ja arvioinnin vaiheet. Tutkimusmenetelmän kehys ohjasi kahden luotavan artefaktin osan rakentamista: case-yritykselle räätälöityä suorituskykybudjettia ohjelmiston suorituskyvyn valvontaan, sekä suorituskykytestejä, joiden tarkoituksena oli testata case-yrityksen ohjelmiston suorituskykyä. Näistä osioista muodostuva artefakti suunniteltiin vastaamaan tutkielmassa tunnistettuihin case-yrityksen tarpeisiin parantaa kehitettävän ohjelmiston suorituskyvyn valvontaa vaihtelevien kuormitustilanteiden yhteydessä. Tutkimusmenetelmän iteratiivisilla suunnittelu- ja validointiprosesseilla pyrittiin varmistamaan, että saavutetut tulokset eivät ainoastaan edistä case-yrityksen käytännön toteutuksia, vaan tuottaisivat myös yleistettävämpää tietämystä erilaisista ohjelmistojen suorituskyvyn testaus-, budjetointi-, sekä valvontatyökaluista sekä strategioista.

Kappaleen ensimmäisessä alaluvussa 6.1 kuvataan suorituskykybudjetin laadintaprosessia sekä sen toteutusta. Alaluvussa kartoitetaan menetelmiä suorituskykybudjetin raja-arvojen määrittelemiselle, sekä määritellään valitut raja-arvot. Tämän lisäksi käydään lyhyesti läpi laaditun suorituskykybudjetin implementointi osaksi luotavia suorituskykytestejä.

Kappaleen toisessa alaluvussa 6.2 suoritetaan teknologiakartoitus, joiden pohjalta suorituskykytestit toteutetaan case-yritykselle. Alaluvussa tunnistetaan tutkimusongelma, jonka pohjalta kartoitustyö aloitetaan. Kartoituksen pohjalle määritellään suorituskykytesteille tavoitteet sekä näiden pohjalta kriteerit valittavalle teknologialle. Tavoitteiden ja kriteerien jälkeen esitellään eri vaihtoehtoja iterointia hyödyntäen, minkä jälkeen palautetta hyödyntäen valitaan sopivin teknologia suorituskykytestien toteuttamiselle. Alaluvun lopuksi kuvataan lopullinen toteutus sekä toteutuksen implementointi osaksi tuotekehitysprosessia

6.1 Suorituskykybudjetin laadinta ja toteutus

Tässä alaluvussa käydään läpi suorituskykybudjetin laadinta sekä perustelut case-yrityksen valitsemille suorituskykybudjetin mittareille. Case-yrityksen suorituskykybudjetin laadinta perustuu kahteen kokonaisuuteen, jotka nousivat esiin case-yrityksen tuotteen testauksen suunnittelussa. Ensimmäinen kokonaisuus on tuotteen eri listausnäkymien nopeus, toinen kokonaisuus on tuotteen isoimpien laskentaominaisuuksien nopeuksien testaus. Näinollen suorituskykybudjetti tullaan rakentamaan näiden kokonaisuuksien seurantaan.

6.1.1 Menetelmät raja-arvojen määrittämiselle

Suorituskykybudjetin raja-arvojen asettamista lähestyttäessä havaittiin kaksi erilaista tapaa, joilla suorituskykybudjetin raja-arvojen määrittämistä voidaan lähestyä: raja-arvot voidaan määrittää suoraan tuotteen suorituskyvyn nykyhetkestä, tai testaamalla sen eri julkaisu- tai päivitysversioita, joista voitaisiin määrittää jonkinlainen keskiarvo asetettaville raja-arvoille. Ensimmäisen menetelmän etuna on menetelmän helppous sekä selkeys. Suorituskyvyn arviointi tulevaisuudessa voi olla yksinkertaisempaa, kun nykyhetkestä valitaan yksi selkeä piste, johon tulevia tuloksia verrataan. Tämä tarkoittaa myös sitä, että testit voidaan lähtökohtaisesti mukauttaa tarkasti nykytilanteeseen sekä tuotteen nykyiseen toimintaympäristöön.

Toisen vaihtoehdon etuina voidaan nähdä historiatiedon kautta saatava parempi kuva tuotteen suorituskyvyn kehittymisestä, sekä antaisi mahdollisuuden tunnistaa kehityksestä hetkiä, jolloin suorituskyky on joko merkittävästi heikentynyt, tai parantunut. Menetelmän uhkana voidaan nähdä, että aiempien tuoteverisoiden pohjalta luotavat raja-arvot voivat antaa vääristyneen kuvan tuotteen nykyhetkestä. Jos historiatietoon perustuvat raja-arvot ovat liian suuret, johtuen esimerkiksi tarkastelujaksolle osuneista hitaammista ohjelmisto- tai tuoteversioista, tällöin suorituskyvyn kehittyminen hidastuu. Syynä on, että suuremmat raja-arvot päästävät läpi muutoksia, jotka voivat jopa heikentää tuotteen nykyistä suorituskykyä.

Näiden havainnointien pohjalta case-yrityksen kehitystiimi päätyi käyttämään menetelmää, jossa se asettaa tuotteelle arvot itse nykyhetken suorituskykyyn verraten. Tämä mahdollistaa nykyhetkessä tarkemman ja ajankohtaisemman suorituskyvyn arvioinnin, joka vastaa nykyhetken tarpeita ja vaatimuksia. Tällä tavoin voidaan myös varmistaa, että tuotteen tai ohjelmiston kehitystä arvioidaan relevantisti ja nykyisistä lähtökohdista käsin, ilman mahdollisten menneiden ja jo korjattujen suorituskykyongelmien vaikutusta nykyhetkeen.

6.1.2 Valitut raja-arvot

Kuten tutkimuksen teoriaosuudessa toteutetussa suorituskykybudjettiin liittyvässä kirjallisuuskatsauksessa havaittiin, on suorituskyvyn budjetointi sekä suorituskykytestaus ylipäätänsä sisällöiltään hyvin tapauskohtaisia. Tällä tarkoitetaan sitä, että testauksessa halutaan testata juuri testauksen kohteena olevaan tuotteeseen liittyviä ominaisuuksia, sekä kehitystiimin tuotteellensa asettamia tavoitteita.

Case-yrityksen kehitystiimi päätyi asettamaan suorituskykybudjetille seuraavat raja-arvot:

- Listausnäkymien latausaika käyttöliittymällä: alle 5 sekuntia
- Budjettiominaisuuden avautuminen käyttöliittymällä: alle 30 sekuntia
- Raportointiominaisuuden avautuminen käyttöliittymällä: alle 10 sekuntia

Kehitystiimi asetti suorituskykybudjetille nämä raja-arvot seuraavin perustein. Listausnäkymille asetettu 5 sekunnin latausaika asetettiin, sillä case-yrityksen ohjelmisto sisältää paljon erilaisia listausnäkymiä, joiden nopeus sekä toimivuus vaikuttaa suoraan tuotetta käyttävän asiakkaan käyttökokemukseen, ja tätä kautta asiakastyytyväisyyteen. 5 sekunnin raja-arvo nähtiin kehitystiimin puolesta sopivana raja-arvona, sillä alle tämän raja-arvon sijoittuva toiminta-aika listaukselle nähtiin riittävän nopeana tuotteen toimintavasteena, kun vastaavasti raja-arvon ylittävä aika on hyvin todennäköisesti merkki jostain tuotteessa piilevästä ongelmasta, joka vaatisi optimointia tai muita korjaustoimenpiteitä.

Suorituskykybudjetin toiseen ja kolmanteen raja-arvoon tuotteen laskentaominaisuuksien suorituskyvyn seurantaan päädyttiin pitkälti samoin perustein. Tuotteen suuret laskentaominaisuudet ovat tuotteen ydin, joiden avulla käsitellään hyvinkin suuria datamääriä sekä monimutkaisia rakenteita. Tästä syystä näiden ominaisuuksien riittävän nopea vasteaika on elintärkeää tuotteen toimivuudelle, käytettävyydelle, sekä asiakastyytyväisyydelle. Edellä mainittujen datamäärien sekä rakenteiden käsittelyn takia kehitystiimi näki 30 sekunnin sekä 10 sekunnin raja-arvon riittäväksi määrittämään näiden ominaisuuksien käytettävyyttä.

6.1.3 Suorituskykybudjetin implementointi

Suorituskykybudjetin raja-arvot implementoidaan osaksi suorituskykytestejä seuraavalla tavalla. Jokaiseen luotavaan, alaluvussa 6.2. esiteltävään, suorituskykytestiin annetaan testattavaa kohdetta vastaava raja-arvo määritellystä suorituskykybudjetista. Tämä raja-arvo kuvaa testin onnistunutta läpäisyä. Mikäli testi onnistuu tavoitteessaan, eli suoriutuu tehtävistään sille annetun raja-arvon puitteissa, on tällöin testi onnistunut ja testattava ominaisuus pysyy sille annetun budjetin rajoissa. Mikäli testi ylittää sille budjetoidun raja-arvon, on testi tällöin epäonnistunut ja testattavaan ominaisuuteen on tehtävä tarvittavat muutokset, jotta sekä yksittäisen ominaisuuden sekä koko testattavan

järjestelmän suorituskyky pysyy sille annetun budjetin rajoissa. Esimerkki suorituskykybudjetin raja-arvojen implementoinnista osaksi suorituskykytestejä on annettu suorituskykytestien esittelyn yhteydessä alaluvussa 6.2.7.

6.2 Suorituskykytestauksen toteutus

Tässä alaluvussa käydään läpi suorituskykytesteihin liittyvä teknologiakartoitus sekä suorituskykytestien toteuttaminen valittua tutkimusmenetelmää hyödyntäen. Ensimmäiseksi tunnistetaan ongelma nykyisessä tuotekehitysprosessissa sekä määritellään tarve suorituskykytestien rakentamiselle. Seuraavaksi määritellään testaukselle asetettavat tavoitteet, joiden perusteella johdetaan valintakriteerit valittaville teknologioille. Tämän jälkeen esitellään kriteerien pohjalta valitut teknologiat ja käydään läpi niistä saatu palaute. Lopuksi palautteen pohjalta valitaan yksi teknologia suorituskykytestien toteutukseen, ja kuvataan testien toteutus, rakenne sekä implementointi osaksi nykyistä tuotekehitysputkea.

6.2.1 Ongelman tunnistaminen

Kohdeyrityksen tuotekehitysprosessissa on tunnistettu nykyisen testausprosessin vajavaisuus havaita tiettyjä tuotteen ominaisuuksia, jotka heikentyessään aiheuttavat mahdollisesti suurtakin haittaa tuotteen loppukäyttäjälle. Kohdeyrityksen tuote pohjautuu erilaisiin laskentoihin sekä niiden esittämiseen. Kohdeyritysten asiakkailla on mahdollista toteuttaa laskentoja suurillakin datamäärillä, minkä lisäksi laskentaan sekä datan esittämiseen on mahdollista vaikuttaa erilaisilla kohdistuksilla, mikä vaikuttaa vaadittavan laskennan kompleksisuuteen.

Ongelmaksi on tunnistettu tällaiseen laskentalogiikkaan kohdistuvan riittävän kattavan testauksen puute. Nykyisessä tuotekehitysprosessissa ja nykyisillä testaustoteutuksilla on mahdollista saavuttaa tilanne, jossa tuotteeseen tehdyn muutoksen mahdollinen negatiivinen vaikutus suorituskykyyn havaitaan vasta siinä vaiheessa, kun tuote on tuotantoympäristössä asiakkaiden käytössä. Tämä johtuu siitä, että nykyisillä testausprosesseilla ei ole käytössään yhtä monipuolista sekä monimutkaista dataa, mitkä tuotantoympäristössä on. Toinen syy on jo toteutetun testauksen keskittyminen tuotteen kokonaistoimivuuteen, eikä sillä testata riittävän laajasti yksittäisiä, tuotteelle ominaisia sekä sen toimille kriittisiä laskentaominaisuuksia.

Ongelman ratkaisemiseksi tässä tutkielmassa määritellään sekä toteutetaan edellä kuvattujen tuotteelle ominaisten sekä kriittisten laskentojen suorituskyvyn testausprosessi.

6.2.2 Suorituskykytestauksen tavoitteet

Määriteltävän suorituskykytestauksen tavoitteena on määrittää testausprosessi tuotteen laskentaominaisuuksien testaamiseen monimutkaisten ja suurien datamäärien tilanteissa. Kappaleessa 3.2 esitellyn taulukon 2 mukaisen luokittelun pohjalta toteutettavista testeistä voidaan puhua sekä kuormitustesteinä että volyymitesteinä, sillä testien tarkoituksena on testata järjestelmän nopeutta sekä laskentakykyä monimutkaisten ja suurien datamäärien avulla.

Testien tavoitteena on yksittäisten laskentaominaisuuksien testaaminen erikseen, jolloin testeistä voidaan puhua yksikkötesteinä, joita on kuvattu alaluvussa 2.4. Alaluvun kuvauksen mukaisesti yksikkötestit keskittyvät yksittäisten komponenttien testaukseen eristetyssä ympäristössä. Toisaalta testit voidaan nähdä myös samassa alaluvussa kuvattuina järjestelmätesteinä, joissa testataan usean eri komponentin toimintaa läpi järjestelmän. Valittu testausteknologia ja sen testauskäytänteet tulevat määrittelemään paljon sitä, mille testaustasolle luotavat suorituskykytestit lokeroidaan. Alaluvun 2.5 mukaisen määrittelyn mukaan luotavat testit voidaan määritellä myös ei-funktionaalisiksi testeiksi, sillä testien tarkoituksena on havainnoida sitä, kuinka hyvin testattava tuote käyttäytyy. Saman alaluvun pohjalta testeistä voidaan puhua ns. White Box-testeistä, sillä testattavan järjestelmän rakenne, koodi sekä arkkitehtuuri on testausvaiheessa tunnettu.

Yllä kuvattujen testien toteuttamiseksi sekä tavoitteiden saavuttamiseksi tulee tuotteesta tunnistaa oikeat komponentit, joita vasten testejä aiotaan suorittaa. Testauksen suunnittelussa on tunnistettu testattaviksi komponenteiksi 19 listauskomponentteja sekä 2 laskentakomponenttia, joita testeillä halutaan havainnoida. Kyseisten komponenttien lisäksi testit vaativat tuekseen oikeanlaisen testausdatan, joka vastaa mahdollisimman hyvin tuotantoympäristössä jo ennestään havaittuja ongelmatapauksia. Näiden määrittelyjen pohjalta on mahdollistaa toteuttaa suorituskykytestit, jotka kattavat riittävän osan testeille osoitetuista tavoitteista testien laajuudella sekä käytössä olevan testausdatan avulla.

6.2.3 Valintakriteerien määrittämien

Valittavien teknologioiden valintakriteereiksi on tutkielmassa määritelty teknologian ominaisuudet sekä soveltuvuus ja integroitavuus nykyiseen tuotekehitysputkeen sekä teknologian mahdolliset kustannukset. Valintakriteerit on laadittu yhdessä case-yrityksen tuotekehitystiimin kanssa. Kriteerien laadinnan tavoitteena on löytää teknologia, joka soveltuisi mahdollisimman suoraviivaisesti case-yrityksen tuotekehitystiimin tarpeisiin eikä vaatisi tuotekehitystiimiltä liian suurta panosta esimerkiksi uuden ohjelmointikielen opiskelussa teknologian käyttöönottoa varten. Valittavan teknologian tulisi myös sisältää sellaisia ominaisuuksia, joilla on mahdollista

mitata suorituskykybudjettiin luvussa 6.1.2 määriteltyjä ohjelmiston ominaisuuksia mahdollisimman helposti sekä riittävän tehokkaasti.

Teknologian tulisi myös soveltua mahdollisimman hyvin jo käytössä oleviin teknologioihin sekä ympäristöihin. Valittava teknologia tulisi oltava integroitavissa mahdollisimman hyvin nykyiseen tuotekehitysputkeen. Case-yritys hyödyntää tuotekehityksessään GitHubia sekä GitHub Actioneita, joten soveltuvuus näihin on erittäin tärkeä valintakriteeri. GitHub on palvelu, jossa ohjelmistokehittäjät voivat tallentaa sekä hallita koodiaan. GitHub perustuu Git-versionhallintajärjestelmään, joka seuraa koodimuutoksia sekä mahdollistaa useiden kehittäjien työskentelyn samassa projektissa. (*About GitHub and Git*, n.d.) GitHub Actions on GitHubin ominaisuus, joka mahdollistaa automaation suoraan GitHub-repositorion sisällä tarjoten mahdollisuuksia erilaisten jatkuvan tuotekehityksen ratkaisujen rakentamiseen. Actioneiden avulla kehittäjät voivat automatisoida esimerkiksi koodin testausta, rakentamista sekä käyttöönottoa tilanteissa, joissa koodiin tehdään jokin muutos. Tämä mahdollistaa sen, että koodi pysyy mahdollisimman korkealaatuisena sekä julkaisuvalmiina, kun kaikki tarvittavat toimenpiteet voidaan automatisoida koodimuutosten yhteyteen. (*GitHub Actions Documentation*, n.d.)

6.2.4 Kartoitetut teknologiavaihtoehdot

Tässä kappaleessa esitellään tutkimuksessa kartoitetut ja case-yritykselle esiteltävät teknologiat, joiden avulla suorituskykytestausta olisi mahdollista suorittaa. Case-yritykselle esiteltäviksi valitut teknologiat olivat Apache JMeter, Azure Load Test sekä Playwright-testikirjasto.

Apache JMeter

Apache JMeter on ilmainen avoimen lähdekoodin ohjelmisto, joka on suunniteltu suorituskyvyn testaukseen sekä kuormitustestaukseen erilaisille verkkopalveluille. JMeter pystyy simuloimaan raskaita käyttäjäkuormia ja mittaamaan, kuinka hyvin erilaiset verkkosovellukset, palvelimet tai tietokantajärjestelmät toimivat näissä olosuhteissa. JMeter on toteutettu Java-ohjelmointikielellä, ja sitä voidaan käyttää kaikissa Java-yhteensopivissa käyttöjärjestelmissä, kuten esimerkiksi Windows-, Linux- tai MacOS-käyttöjärjestelmissä. JMeter hyödyntää graafista käyttöliittymää, mikä helpottaa monimutkaisempien testien suunnittelua. Lisäksi se tukee erilaisia laajennuksia sekä skriptejä, joiden avulla on mahdollista räätälöidä testejä vastaamaan erityisiä testausvaatimuksia. JMeterin ominaisuuksiin kuuluu HTTP-, HTTPS-, SOAP- ja REST-verkkopalveluiden testaus, FTP-palveluiden testaus, tietokantakyselyiden testaus sekä paljon muuta. JMeter tukee myös rinnakkaistestien suorittamista, mikä mahdollistaa laajempien sekä monimutkaisempien kuormitustestien suorittamisen. (*Apache JMeter - User's Manual*, n.d.)

Apache JMeter on noussut viime vuosina yhdeksi suosituimmaksi kuormitustestaustyökaluksi. Suosion kasvuun ovat vaikuttaneet JMeterin avoimen lähdekoodin luonne, laaja ekosysteemi, sekä kyky houkutella kehittäjiä jatkokehittämään tuotetta. Lisäksi JMeterin skripteistä on tullut de facto-standardi, ja monet SaaS-työkalut tukevat niitä, mikä laajentaa JMeterin toiminnallisuutta ja tekee siitä kilpailukykyisen myös kaupallisten vaihtoehtojen rinnalla. (Rodrigues et al., 2019)

Apache JMeterin valintaa esiteltäväksi teknologiavaihtoehdoksi puoltaa erityisesti sen todistettu soveltuvuus testata ohjelmiston suorituskykyä. Toiseksi JMeterin laaja ekosysteemi dokumentaatioineen sekä kehittäjien osallistuminen tuotteen ylläpitoon varmistaa sen, että ratkaisulla on mahdollista toteuttaa hyvin erilaisia testauskokonaisuuksia ja niille on olemassa tuki myös jatkossa. Kolmas valintaa puoltava tekijä on se, että ohjelmisto on saatavilla ilmaiseksi.

Azure Load Test

Azure Load Testing on Microsoftin tarjoama pilvipohjainen palvelu, joka on suunniteltu sovellusten suorituskyvyn testaamiseen todellisia käyttäjäkuormia simuloimalla. Palvelu mahdollistaa käyttäjille erilaisten kuormitustestien suorittamisen ilman tarvetta oman testausinfrastruktuurin ylläpitämiseen. Palvelu tarjoaa sovellusten reaaliaikaista suorituskyvyn seurantaa sekä analytiikkaa, joiden avulla kehittäjien on mahdollista tunnistaa pullonkauloja ja tätä kautta optimoida sovellusten suorituskykyä. Azure Load Testing on integroitavissa muiden Azure-palveluiden kanssa, ja se pystyy hyödyntämään esimerkiksi Apache JMeter-kirjastoa. (*What Is Azure Load Testing?*, 2023)

Microsoft Azure on Microsoftin kehittämä pilvipalvelualusta, joka tarjoaa yli 200 tuotetta tai palvelua sovellusten rakentamiseen, käyttöönottoon sekä hallintaan. Azure tukee monia eri käyttöympäristöjä, kuten Infrastructure as s Service (IaaS), Platform as a Service (PaaS) sekä Software as a Service (SaaS). Azure tarjoaa ratkaisujaan pilvi- ja paikallisympäristöille, sekä näiden hybridimalleille. (*What Is Azure – Microsoft Cloud Services | Microsoft Azure*, n.d.)

Azure Load Testing-palvelun käyttöönottoa puoltaa se, että case-yritys hyödyntää jo ennestään Azuren tarjoamia palveluja. Tätä kautta testien integrointi eri Azure-ympäristöihin olisi mahdollista kyseisellä tuotteella. Toisena tekijänä on palvelun tarjoamat työkalut suorituskyvyn reaaliaikaiseen seurantaan sekä testaustulosten analysointiin. Kolmantena tekijänä voidaan nähdä edellä esitellyn Apache JMeterin skriptien soveltuvuus palvelun kanssa. Tämä mahdollistaisi kehittäjälle JMeterin käyttöliittymän hyödyntämisen testien suunnittelussa, jolloin Azure Load Testing-palvelu toimisi testien suorittajana, sekä seuranta- ja analysointityökaluna.

Playwright

Playwright on Microsoftin kehittämä avoimen lähdekoodin testikirjasto, joka on suunniteltu tarjoamaan kehittäjille kattavat työkalut verkkosovellusten automatisoituun testaukseen. Playwright tukee useita selainmoottoreita, kuten

Chromiumia, Firefoxia sekä WebKitiä, ja mahdollistaa näin samanaikaisen testauksen eri alustoilla. Kirjasto tarjoaa moderneja testausominaisuuksia, kuten esimerkiksi täyden tuen JavaScriptin, TypeScriptin sekä muiden modernien teknologioiden käytölle. Kirjasto mahdollistaa kehittäjien luoda ja suorittaa erilaisia end-to-end-testejä, joilla simuloidaan käyttäjien vuorovaikutusta verkkosovellusten kanssa. (*Playwright*, n.d.)

End-to-end (E2E) -testausmenetelmällä pyritään havainnoimaan sitä, että testattavan järjestelmän toiminnot toimivat oikein alusta loppuun. Testauksessa jäljitellään todellisia käyttäjäskenaarioita, jolloin testauskohteena on koko järjestelmän työnkulku, missä yhdistyvät esimerkiksi tietokannat, sovellusrajapinnat sekä käyttöliittymä. Tavoitteena on varmistua siitä, että järjestelmä käyttäytyy odotetusti mahdollisimman realistisissa olosuhteissa, eikä virheitä esiinny missään skenaarion vaiheessa. (Hernández ym., 2021)

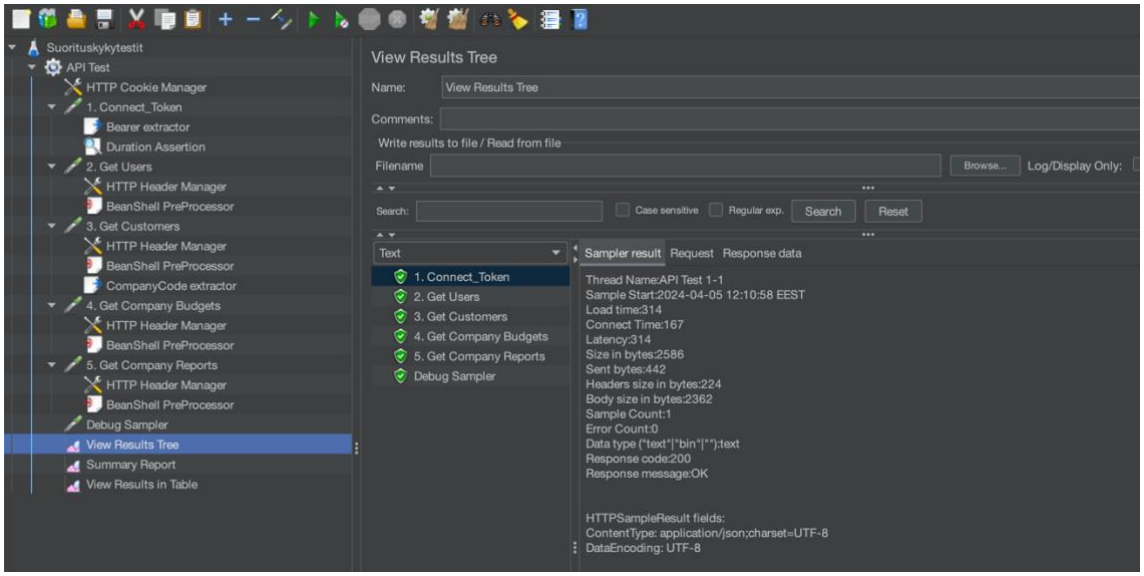
Playwrightin käyttöönottoa case-yrityksen kohdalla tukee se, että sitä on tutkimuksen tekovaiheessa otettu jo asteittain käyttöön uutena teknologiana jo olemassa olevien käyttöliittymää testaavien end-to-end-testien korvaajana. Hyödyntämällä samaa kirjastoa myös luotaviin suorituskykytesteihin helpottaisi valinta suorituskykytestauksen ylläpitoa sekä kehittämistä, kun samoja teknologioita on käytössä läpi case-yrityksen tuotteen. Kirjaston käyttöä puoltaa myös sen modernit teknologiaratkaisut, sekä ilmaisuus.

6.2.5 Vaihtoehtojen esittely

Tässä alaluvussa kuvataan edellisessä kappaleessa esiteltyjen teknologiavaihtoehtojen esittely case-yritykselle tuotekehitystiimille. Esittely sisälsi esimerkkitestit jokaista esiteltävää teknologiaa hyödyntäen, sekä kuvaus niiden implementoinnista nykyisiin tuotekehitysprosesseihin. Tässä luvussa käydään lyhyesti läpi esiteltävät esimerkit jokaisesta teknologiasta.

Apache JMeter

Ensimmäisenä esiteltävänä teknologiana oli Apache JMeter. Esittelyä varten luotiin JMeterin käyttöliittymällä yksinkertainen testitapaus, joka on esitelty kuvassa 1. Kuvan mukaisesti testissä toteutetaan viisi (5) kappaletta yksinkertaisia kutsuja, jotka kohdistettiin suoraan järjestelmän backend:iin. Testissä ei näinollen huomioitu lainkaan järjestelmän käyttöliittymää. Testissä suoritetaan ensin käyttäjän autentikointi, minkä jälkeen suoritetaan neljä yksinkertaista GET-kutsua hakemaan erilaisia resursseja. Kaikille kutsuille asetettiin suorituskykybudjetissa määritelty raja-arvo. Raja-arvon alittaminen kertoi onnistuneesta ja ylitys vastaavasti epäonnistuneesta testistä.

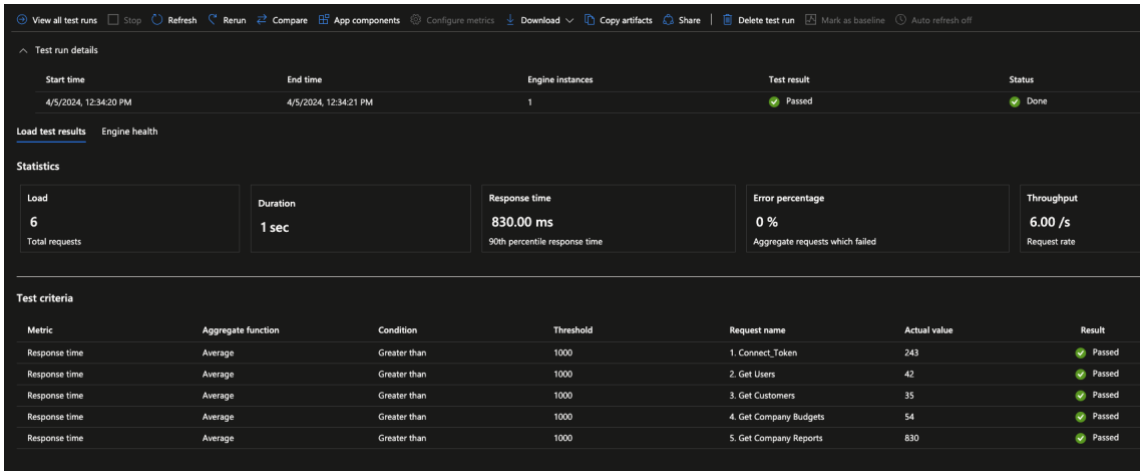


Kuva 1. Apache JMeter esimerkkitestit

Seuraavaksi kuvattiin testien ajaminen JMeterillä osana GitHub Actioneitä. Testit ajettiin pull-requestin sisällä Actioneita hyödyntäen. Esimerkkitestejä rakennettaessa ei löydetty valmista toimivaa kirjastoa testien ajamiseen, joten testien ajaminen sekä tulosten esittäminen jouduttiin rakentamaan itse käsin komentorivikomennoilla. Tämä indikoi JMeterin käytön kompleksisuutta sekä mahdollista kehittämistä tulevaisuudessa, mikäli toimivia kirjastoja ei olisi saatavilla ja mahdolliset muutokset tulisi kehittäjien itse rakentaa.

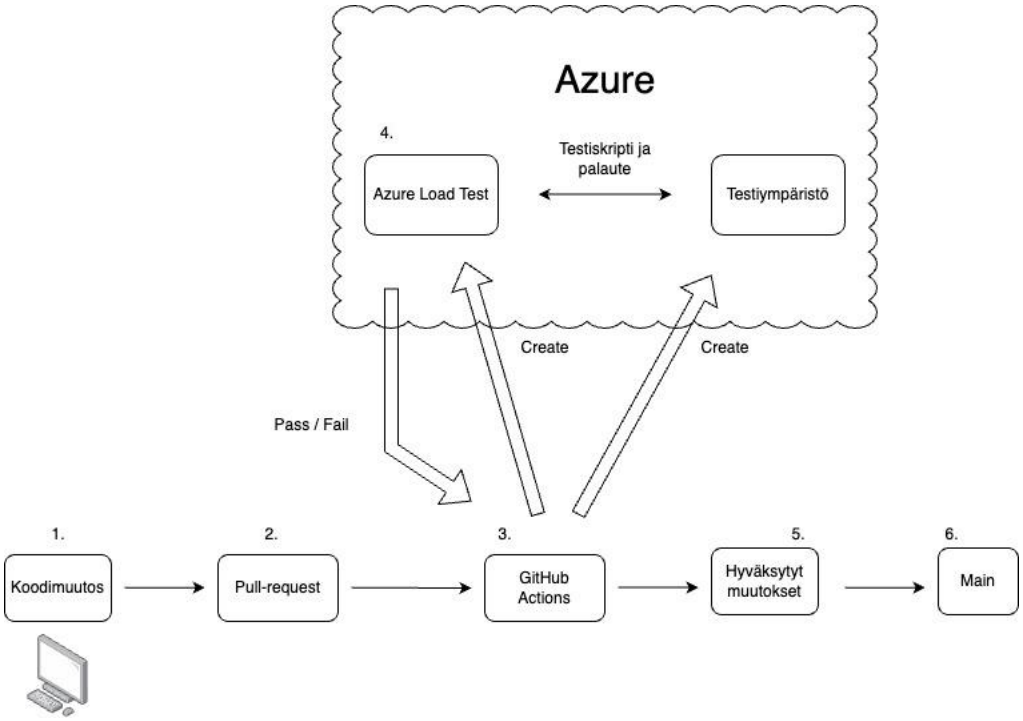
Azure Load Test

Toisena teknologiana esiteltiin Azure Load Test. Esittelyä varten luotiin yrityksen yhteen Azure-ympäristöön Azure Load Test- testiympäristö, jonne ladattiin ajattavaksi testiskriptiksi edellisessä osassa luotu Apache JMeter-testiskripti. Testin rakenne sekä raja-arvot pysyivät siten identtisinä. Testeistä saadut tulokset Azure Load Test-ympäristössä on esitelty kuvassa 2. Kuvassa on ajettu edellä kuvattu Apche JMeter-testiskripti yhtä case-yrityksen Azure:ssa olevaa ympäristöä vasten, ja tulokset sekä ajon muu analytiikka on kuvattu testiajon päätteeksi.



Kuva 2. Azure Load Test-ympäristön tulokset

Koska testien ajaminen Azure Load Test-ympäristöllä eroaa kahedsta muusta valitusta testausvaihtoehdosta, joissa testit tullaan suorittamaan GitHub Actionin sisällä, on tästä syystä testien ajaminen kuvattu kuviossa 1. Kuten kuviosta voidaan havaita, testien suorittaminen tapahtuu GitHub actionin ulkopuolella erillisessä Azure-ympäristössä. Koodimuutosten myötä testiaction luo kohdassa 3 ensin Azureen testiympäristön, jossa tehdyt koodimuutokset ovat mukana. Tämän jälkeen kohdassa 4 action käynnistää annetun testiskriptin Azure Load Test-ympäristössä, joka ajaa testit osoitettua luotua testiympäristöä vasten. Ajon jälkeen testien tulokset esitetään Azuressa, ja tiedot testien tuloksista lähetetään takaisin pull-request:iin.



Kuvio 1. Azure Load Test osana GitHub Actioneitä

Playwright

Kolmantena teknologiana esiteltiin Playwright. Esittelyä varten luotiin vastaava testitapaus kuten aiemmissa esimerkeissä. Erona edellisiin testiskripteihin oli, että Playwrightin tapauksessa kieli muuttui Javasta JavaScriptiin, ja testit toteutettiin käyttäen Playwrightin omia metodeja. Esitellyt esimerkkitestit ovat listattuna kuvassa 3., ja ne vastaavat aiempien esiteltyjen vaihtoehtojen testitapauksia, missä suoritetaan yksinkertaisia näkymien latauskutsuja. Playwrightin testeillä simuloidaan haettavien resurssien GET-kutsuja käyttöliittymän kautta. Näiden kustujen valmistumiselle on asetettu suorituskykybudjetissa määritelty timeout-arvo, mikä kuvaa testien onnistumista ja järjestelmän osan kykyä pysyä annetussa budjetissa.

```
7   import { test } from './fixtures';
8
9   const TIMEOUT = 5000;
10
11  test('test', async ({ page, customer, company }) => {
12    await page.goto(`/customers`,  { timeout: TIMEOUT })
13    await page.goto(`/${customer}/${company}/users`,  { timeout: TIMEOUT })
14    await page.goto(`/${customer}/${company}/budgets`,  { timeout: TIMEOUT })
15    await page.goto(`/${customer}/${company}/reports`,  { timeout: TIMEOUT })
16  });
17
```

Kuva 3. Playwright-kirjaston esimerkkitestit

Kuten Apache JMeterin esimerkissä, koodimuutokset käynnistävät GitHubissa actionin, joka asentaa sekä rakentaa tuotteen actionin sisäiseen ympäristöön. Tämän jälkeen annetut Playwright-testit ajetaan tämän actionin sisällä kyseistä rakennettua tuoteversiota vasten. Actionin status kertoo ajon onnistumisesta tai epäonnistumisesta, minkä lisäksi testiajosta luodaan raportti erilliseen url-osoitteeseen, joka lisätään koodimuutosten pull request:iin. Sekä testien ajolle että tulosten raportointiin löytyi valmiit työkalut, eikä työkaluja tarvinnut rakentaa itse käsin, kuten Apache JMeterin tapauksessa.

6.2.6 Teknologian valinta ja perustelut

Tässä alaluvussa käydään läpi käytettävän teknologian valinta sekä valinnan perusteet. Edellisen luvun esittelyjen pohjalta kehitystiimi päätyi valitsemaan suorituskykytestaukseen käytettäväksi teknologiaksi Playwright-kirjaston.

Azure Load Test:in jättäminen valinnan ulkopuolelle perusteltiin seuraavasti. Pääasiallisena syynä kehitystiimi näki testauksen rakenteeseen liittyvät ongelmat. Kehitystiimin nykyinen tuotekehitysputki on rakennettu GitHubin ympäristöihin, joissa Actionit ovat suuri osa tätä putkea. Aikaisemmat testausprosessit on pystytty toteuttamaan aina yksittäisen pull-request:in sekä Actionin sisällä, ja tästä haluttiin pitää kiinni myös suorituskykytestauksen

implementoinnissa. Azure Load Test vaatisi Actionin ulkopuolisen resurssin pystyttämistä, jota vasten testejä voitaisiin ajaa. Kehitystiimi koki tämän vaiheen turhana, sillä muutokset halutaan testata jo ennen, kun niiden pohjalta aletaan rakentamaan ympäristöjä. Tästä syystä Azure Load Test jätettiin valinnan ulkopuolelle.

Apache JMeterin jätettiin valinnan ulkopuolelle seuraavin perustein. Toisin kuten Azure Load Testi:in tapauksessa, JMeterillä suoritettava testaus on mahdollista suorittaa yhden pull-requestin sisällä tarkoittaen, että se ei vaadi Actioneiden ulkopuolisten resurssien pystyttämistä testausta varten. JMeter soveltuu myös ominaisuuksiltaan hyvin suorituskyvyn testaukseen, kuten teknologiaa esittelevässä luvussa on kuvattu. Päätökseen jättää teknologia valinnan ulkopuolelle vaikutti eniten se, että kyseessä on erilainen teknologia sekä eri ohjelmointikieli, mitä kehitystiimin nykyisissä ratkaisuissa on käytössä. JMeter pohjautuu Javaan, kun taas vastaavasti tuotteen ohjelmointikieliin lukeutuvat C# sekä JavaScript. Tästä syystä tuotekehitystiimi näki, että uuden ohjelmointikielen käyttöönotto toisi turhaa ylimääräistä ylläpitotyötä, jos vastaavat ominaisuudet voidaan toteuttaa jollain muulla työkalulla, joka hyödyntää tuotekehityksellä jo käytössä olevia ohjelmointikieliä.

Playwright-kirjaston valinta käytettäväksi teknologiaksi perusteltiin seuraavasti. Ensimmäisenä valintaperusteena oli aikaisempi tuntemus teknologiasta. Kyseistä testikirjastoa on otettu käyttöön tuotekehityksessä jo muissa testaustarpeissa, joten sen käyttöönotto myös suorituskyvyn testauksen suorittavana teknologiana koettiin luontevaksi. Kirjasto hyödyntää myös ohjelmointikielenään JavaScriptiä, mikä on tuotteessa käytössä jo entuudestaan. Toinen valintaperustelu oli olemassa olevat ratkaisut testauksen implementointiin osaksi tuotekehitysputkea. Kuten aiemmin mainittiin, kirjastoa on alettu käyttämään tuotekehityksessä jo enne tutkimusta, joten sille on löydetty todistettavasti toimivia ratkaisuja, joiden avulla testausta voidaan suorittaa yksittäisten pull-request:ien sisällä. Kolmantena perusteluna ovat kirjaston ominaisuudet, joilla voidaan rakentaa halutut testitapaukset, joilla on mahdollista mitata alaluvussa 6.1.2 käsiteltyjä suorituskykybudjetin asettamia mittareita. Neljäntenä perusteluna toimi testauksen laatu ja siitä tavoitellut hyödyt. Luotavien suorituskykytestien halutaan edustavan mahdollisimman aitoa käyttötilannetta riittävän rasituksen alla. Tällöin absoluuttisen suorituskyvyn mittaaminen ei ole testattavan tuotteen tapauksessa tärkein prioriteetti, vaan eniten hyötyä saadaan, kun mitataan suorituskykyä aidoissa käyttötilanteissa, aidon rasituksen alla.

6.2.7 Suorituskykytestauksen toteutus

Tässä alaluvussa kuvataan suorituskykytestauksen toteutus käyttäen edellisessä luvussa valittua testausteknologiaa. Luvussa käydään läpi testien toteutusta kyseisellä teknologialla, mitä esitietovaatimuksia testien ajamiseen tarvitaan, ja kuinka testit toteutetaan mittaamaan aiempana määriteltyjä

suorituskykybudjetin raja-arvoja. Luvussa esitellään myös luotujen testien rooli tuotekehitysputkessa.

Testien kirjoittaminen Playwright-kirjastolla tapahtuu kuvan 4. mukaisesti. Testitapaus lähtee liikkeelle ympäristön määrittelystä, jonka jälkeen työkalu avaa sille annetun näkymän ja toteuttaa sille annettuja toimintoja ympäristössä. Kuvan 4. esimerkissä testitapaus avaa rivillä 8 sille annetun url-osoitteen, ja odottaa rivillä 9 sivun sisältävän sisältöä, joka testille on annettu ”toContain”-metodissa. Testin onnistunut ajo edellyttää kaikkien testissä annettujen komentojen suorittamista hyväksytysti. Esimerkin kaltainen yksittäinen Playwright-testin tarkoitus on siis simuloida käyttäjän toimintoja annetussa ympäristössä automatisoidusti, ja näin varmistaa kyseisen ympäristön toimivuus.

```
4
5   const { test, expect } = require('@playwright/test');
6
7   test('testaa Google-haku', async ({ page }) => {
8     await page.goto('https://www.google.com');
9     expect(title).toContain('Google');
10  });
```

Kuva 4. Yksinkertainen Playwright-testi

Ympäristön määrittelyyn sekä testiresurssien alustamiseen sekä jakamiseen testien kesken käytetään Playwrightissa erilaisia fixtureja (*Playwright*, n.d.). Fixturejen avulla voidaan hallita esimerkiksi ympäristön käyttäjien tilaa, käytetyn tietokannan tilaa tai muita testeissä toistuvia toimintoja. Kuvassa 5. on esimerkkitesti, jossa on käytössä kuvassa 6. alustettava tietokantayhteys. Kuvassa 6. luodaan databaseFixture-niminen fixture, joka suorittaa rivillä 18 tietokantayhteyden avaamisen ja odottaa tämän jälkeen kyseisen tietokantayhteyden käyttöä kuvan 5. testiltä. Kun testi on ajettu, fixture sulkee tietokantayhteyden kuvan 6. rivillä 20. Fixturet ovat keskeinen osa Playwright-kirjastolla ajettavia testejä, sekä tässä tutkimustyössä luotavia suorituskykytestejä, sillä määriteltävät testit vaativat toimiakseen ennalta määritellyn ympäristön sekä datan, jotta testattavat ominaisuudet olisivat mahdollisimman edustavia suorituskyvyn testauksen näkökulmasta.

```
7   import { test } from './fixtures';
8
9   test('passes', async ({ database, page, a11y }) => {
10    // database on testin käytössä erilaisiin tietokantakyselyihin
11  });
```

Kuva 5. Fixturejen käyttö Playwright-testeissä

```
15  // fixtures.js
16
17  const databaseFixture = async ({}, use) => {
18    const database = await connectToDatabase(); // Yhteys tietokantaan
19    await use(database); // Annetaan testin käyttää database-resurssia
20    await database.close(); // Siivotaan tietokantayhteys testin jälkeen
21  };
```

Kuva 6. Fixturejen luonti Playwright-kirjastolla

Tässä työssä varsinaisia testitapausten tyyppejä on kaksi suorituskykybudjetin määritelmän mukaisesti: erilaisten listausnäkymien testaus sekä laskentaominaisuuksien testaus ison datamäärän kanssa. Seuraavaksi esitellään esimerkit molemmista luoduista testitapauksista. Esimerkki listausnäkymän testauksesta on kuvattu kuvassa 7. Rivillä 29 alkavassa testissä on tarkoituksena testata listausnäkymää, jonka tehtävänä on näyttää yrityksen kaikki tilit. Testi hyödyntää alustettavana fixtureinaan customer- ja company-fixtureja, joiden tehtävänä on alustaa ympäristön konteksti vastaavaksi mitä se olisi oikean asiakasyrityksen näkökulmasta. Lisäksi testi käyttää listingAccounts-fixturea, jonka luominen on kuvattu kuvassa 8. Kyseisen fixturen avulla testiympäristöön luodaan vaadittava määrä dataa, jotta listausnäkymän suorituskyvyn testaaminen antaisi parhaan palautteen näkymän suorituskyvystä. Kuvan 7 testi navigoi ensin itsensä haluttuun näkymään, jonka jälkeen se jää odottamaan tilin ilmestymistä listaan rivillä 32 alkavassa expect-metodissa. Tälle metodille on rivillä 27 annettu timeout-arvo, joka on suorituskykybudjetin mukaisesti 5 sekuntia. Jos metodi ei toteudu annetun arvon puitteissa, näytetään testi epäonnistuneena ajona. Tämän lisäksi näkymässä testataan jokaisesta listausnäkymästä löytyvää hakutoimintoa riveillä 35 ja 36.

```
25  import { test, expect } from '../../fixtures/perf-test-fixtures'
26
27  const TIMEOUT = 5000
28
29  test('Account listing load test', async ({ page, customer, company, listingAccounts }) => {
30      await page.goto(`/${customer}/${company}/accounts`)
31      await expect(page.getByRole('navigation')).toBeVisible()
32      await expect(
33        page.getByRole('cell', { name: 'Playwright Account 1000', exact: true })
34      ).toHaveCount(1, { timeout: TIMEOUT })
35      await page.fill('input', 'Playwright Account 10993', { timeout: 500 })
36      await expect(
37        page.getByRole('cell', {
38          name: 'Playwright Account 10993',
39          exact: true,
40        })
41      ).toHaveCount(1, { timeout: 500 })
42    }
43  )
```

Kuva 7. Esimerkki listausnäkymän suorituskykytestistä

```
54  //perf-test-fixtures.js
55
56  import perfTestAccounts from 'perfTestAccounts.json'
57
58  listingAccounts: [
59    async ({ company }, use) => {
60      const accounts = await apiRequest({
61        method: 'post',
62        url: company.code + '/accounts/overwrite',
63        body: perfTestAccounts,
64      })
65      expect(accounts).toBeDefined()
66      await use(accounts)
67    },
68    { scope: 'test', timeout: 240000 },
69  ]
70
```

Kuva 8. Esimerkki listausnäkymän fixtureista

Seuraavaksi esitellään esimerkki laskentaominaisuuden suorituskykytestistä. Kyseessä on raportointiominaisuuden testi, mikä on kuvattuna kuvassa 9. Testi on rakenteeltaan hyvin samanlainen kuin aiemmin esitelty listausnäkymän testi. Testi hyödyntää ympäristön alustuksessa eri fixtureja, ja testin läpimeno määritellään testille suorituskykybudjetissa määritellyllä raja-arvon alittamisella. Erona listausnäkymien testiin kyseisessä laskentaominaisuuksien testissä on fixtureiden koko. Kyseisessä testissä fixturet ”bigDataReport” sekä ”bigDataVouchers” luovat testiä varten riittävän monimutkaisen raportin, sekä raportille riittävän suuren tositemäärän. Kun ympäristö on alustettu riittävällä datamäärällä, testi testaa raportointiominaisuuden suorituskykyä avaamalla rivillä 78 fixtureissa luodun raportin. Tämän jälkeen rivillä 79 testi odottaa kyseisen näkymän aukeavan annetun timeout-arvon sisällä. Kuten aiemmissa testiesimerkeissä, annetun raja-arvon alittaminen kertoo onnistuneesta testiajosta eli siitä, että nykyiset koodimuutokset eivät tule aiheuttamaan testatun laskentaominaisuuden tapauksessa suorituskyvyn heikkenemistä.

```
73  import { test, expect } from '../../fixtures/perf-test-fixtures'
74
75  test('Report', async ({ page, customer, company, bigDataReport, bigDataVouchers }) => {
76    await page.goto(`/${customer}/${company}/reports`)
77    await expect(page.getByText('Big Data Report')).toBeVisible()
78    await page.getByRole('link', { name: 'Big Data Report' }).click()
79    await expect(page.getByText('3000 Myynti')).toHaveCount(1, {
80      timeout: 10000,
81    })
82  })
```

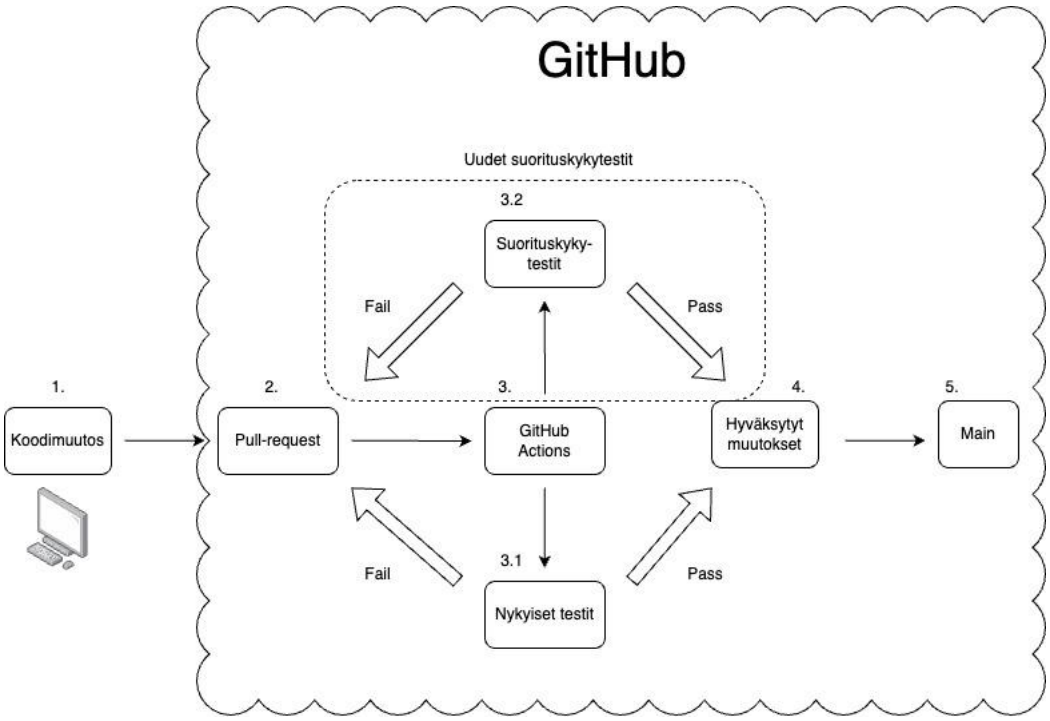
Kuva 9. Esimerkki laskentaominaisuuden suorituskykytestistä

6.2.8 Suorituskykytestit osana tuotekehitysputkea

Tässä luvussa kuvataan edellisessä luvussa esiteltyjen suorituskykytestien rooli osana tuotteen nykyistä tuotekehitysprosessia. Suorituskykytestien rooli on kuvattuna kuviossa 2, missä kuvataan koodimuutoksen reitti osaksi varsianista tuotetta. Kuvan kohdassa 1. kehittäjä rakentaa uuden ominaisuuden tai tekee muutoksia olemassa olevaan koodin. Tämän jälkeen kehittäjä puskee nämä muutokset tuotteen GitHub-ympäristöön, jonne avataan tällöin uusi pull-request kohdan 2. mukaisesti.

Kun uusi pull-request on avattu, siirrytään kohtaan 3. jossa uusia koodimuutoksia testataan automaattisesti GitHub Actions-työkalun avulla. Tässä vaiheessa uudet suorituskykytestit tulevat osaksi tuotekehitysputkea. GitHub Actions ajaa sekä nykyiset, kohdassa 3.1 kuvatut, ennestään olemassa olevat testit, sekä uudet kohdassa 3.2 kuvatut suorituskykytestit. Kuten nykyistenkin testien kohdalla, suorituskykytestien onnistunut ajo määrittää, kuinka muutokset etenevät tuotekehitysputkessa. Jos suorituskykytestit eivät mene läpi, eli tuotteen suorituskyky tulisi nykyisillä muutoksilla laskemaan alle annettujen raja-arvojen, siirryttäisiin kuvan mukaisesti kohtaan 2. Tällöin kehittäjän tulee tehdä tarvittavat muutokset testeistä saatujen palautteiden perusteella, minkä jälkeen muutosten puskeminen pull-requestiin käynnistää jälleen kuvan kohdan 3 automaattiset tarkistukset.

Jos testit menevät läpi, siirrytään tällöin kehitysputkessa kohtaan 4. Tällöin automaattitestien jälkeen voidaan suorittaa muita tarkistuksia muutoksille, kuten esimerkiksi koodikatselmointia toisen kehittäjän toimesta. Jos korjattavaa kohdetta ei tässä vaiheessa löydy, siirrytään kuvassa kohtaan 5, jolloin testatut tuotteen yhdistetään osaksi varsinaista tuotetta.



Kuvio 2. Suorituskykytestit osana tuotekehitysputkea

7 Pohdinta

Tässä kappaleessa käydään läpi tämän tutkimuksen keskeisimpiä havaintoja sekä pohditaan niiden merkitystä sekä teoreettisesta että käytännön näkökulmasta. Tämän lisäksi arvioidaan käytettyjen tutkimusmenetelmien laatua, sekä nostetaan esiin mahdollisia jatkotutkimusaiheita. Tutkimus keskittyi sekä suorituskykytestauksen että suorituskykybudjetin määrittämiseen ja toteuttamiseen tutkimuksen case-yritykselle. Tutkimuksen merkitys korostui erityisesti ohjelmiston kehityksen sekä ylläpidon näkökulmasta, sillä kuten tutkimuksessa nousi esiin, on suorituskyvyn hallinta sekä optimointi kriittisiä tekijöitä ohjelmiston laadun sekä parhaan mahdollisen käyttökokemuksen varmistamisessa.

7.1 Teoreettinen kontribuutio

Tutkimuksen teoreettisena kontribuutiona voidaan pitää tutkimuksessa toteutettua kirjallisuuskatsausta, jossa nostettiin esiin suorituskykybudjettia käsitteenä sekä niitä tekijöitä, mitkä vaikuttavat suorituskykybudjetin laadintaan ja sen sisältöön.

Kirjallisuuskatsauksessa nousi esiin suorituskykybudjetoinnin rooli osana modernia ohjelmistokehitystä erityisesti silloin, kun kehitettävässä ohjelmistossa halutaan tavoitella sekä ylläpitää tasapainoa suorituskyvyn ja käyttäjäkokemuksen välillä. Muun suorituskykytestaukseen liittyvän teorian kanssa tutkimus vahvistaa käsitystä siitä, että suorituskykytestauksen ja budjetoinnin yhteisvaikutuksella voidaan tunnistaa kriittisiä suorituskyvyn rajoja, jotka ohjelmiston tulisi täyttää. Tällainen suorituskyvyn budjetointi auttaa optimoimaan järjestelmän toimintaa sekä estää suorituskyvyn heikkenemisen ohjelmiston päivitysten tai muutosten yhteydessä.

Suorituskykybudjetin laatiminen on monivaiheinen prosessi, jonka onnistuminen edellyttää monien eri näkökulmien perusteellista pohdintaa. Budjetin oikeellisuuden varmistamiseksi on tärkeää, että budjetti perustuu sekä ohjelmiston käyttötarkoitukseen että aitoihin käyttäjävaatimuksiin. Riittävän varmuuden saavuttaminen edellyttää käyttäjäpalautteen, käyttäjätarinoiden

sekä suorituskykytestien yhdistämistä ja tätä kautta jatkuvaa testausta. Budjetin tulisi olla myös dynaaminen, eli sitä tulisi voida tarkentaa ja päivittää aina tarpeen mukaan, jotta se vastaa testattavan ohjelmiston kehittyviä tarpeita sekä käyttäjäkokemuksia. Vaikka ohjelmisto pysyy määrätyn suorituskykybudjetin rajoissa, voidaan tällöin esittää kysymys, ovatko asetetut raja-arvot liian suuria? Tällaisissa tilanteissa suorituskykyä ei välttämättä olla optimoitu tarpeeksi, ja budjetti voi olla liian löyhä suhteessa siihen, mitä olisi mahdollisesti saavutettavissa. Toisaalta, jos suorituskyky ei pysy budjetin rajoissa, voi syynä olla selkeä puute suorituskyvyssä tai budjetin liian tiukat raja-arvot. Tästä syystä dynaamisuus on tärkeä osa suorituskykybudjettia.

Suorituskykybudjetin määrittelyssä on myös hyvä tunnistaa suhde ohjelmiston absoluuttisen suorituskyvyn sekä käyttökokemuksen välillä. Ohjelmiston käytön sujuvuus sekä käyttäjätyytyväisyys voivat olla kaupallisessa tuotteessa monelle loppukäyttäjälle suurempi kriteeri kuin teknisesti täydellinen suorituskyky. Kumpi on tärkeämpää, riippuu lopulta itse ohjelmiston käyttötarkoituksesta. Tutkimuksen case-yrityksen tapauksessa ohjelmiston käyttökokemuksella on erittäin suuri merkitys, jolloin ei välttämättä ole kannattavaa tavoitella ohjelmiston absoluuttista suorituskykyä käyttökokemuksen kustannuksella.

Tutkimuksen kirjallisuuskatsaus osoitti, että suorituskykybudjetin teoria sekä suorituskykybudjetti käsitteenä ovat vielä suhteellisen uusi alue. Toteuttamalla uusia tutkimuksia kirjallisuuskatsauksessa esiin nousseisiin suorituskykybudjetin osa-alueisiin on mahdollista saavuttaa laajempaa ymmärrystä siitä, mitä kaikkea suorituskykybudjetin laadintaan ja käyttöön on tarpeellista sisällyttää.

7.2 Käytännön kontribuutio

Tutkimuksen käytännön kontribuutiot keskittyvät ensisijaisesti case-yritykselle luotuun suorituskykybudjettiin ja sen budjetointiprosessiin, sekä itse suorituskykytesteihin. Näitä konkreettisia tuloksia voidaan kuitenkin soveltaa yleisesti suorituskyvyn budjetoinnissa sekä testauksessa jatkuvissa ohjelmistokehitysprosesseissa.

Case-yrityksen osalta tutkimuksesta saavutettu käytännön kontribuutio tarkoitti, että case-yritys sai tutkimuksen kautta luotua itselleen tavan budjetoida sekä valvoa oman ohjelmistonsa suorituskykyä tehokkaammalla tavalla. Kuten tutkimuksessa nousi esiin, sekä tapoja määrittää eri budjetteja sekä ylipäätänsä haluttua suorituskykyä on yhtä monta kuin on testattavaa ohjelmistoakin. Näin ollen käytännön kontribuutio suorituskykybudjetin osalta case-yritykselle onkin tarjota alkupiste budjetoinnille sekä keinoja budjetin jatkojalostamiseen tulevaisuudessa. Kun case-yritykselle nousee tarve määrittää budjettia uudestaan, tai tarve laatia se kokonaan uudestaan, voi se hyödyntää tutkimuksessa esiin nousseita näkökulmia ikään kuin askelmerkkeinä

määrittämään, mitä kaikkea suorituskykybudjetin määrittämisessä kannattaa ottaa huomioon.

Toinen selkeä tutkimuksella tuotettu käytännön kontribuutio on case-yritykselle luodut suorituskykytestit. Tässäkin tapauksessa tutkimus toimii lähtökotana case-yrityksen suorituskykytestaukselle, tarjoten sille vaihtoehtoja olemassa olevien testausvaihtoehtojen sekä nykyisen tuotekehitysympäristön soveltuvuudesta. Kun lähtökohta suorituskykytestaukselle on tutkimuksella saavutettu, voidaan case-yrityksessä paremmin arvioida sitä, ovatko nykyiset vaihtoehdot riittäviä vai tulisiko testausta jollain tapaa muokata tulevaisuudessa. Kuten suorituskyvyn budjetoinnin, myös suorituskykytestauksen osalta on tärkeää löytää tasapaino testauksen sekä siitä saavutettavien hyötyjen välillä. Testauksen tulisi olla riittävän kattavaa, mutta sen ei tulisi haitata ohjelmiston kehitystyötä liiaksi. On myös tärkeää pohtia sitä, missä suhteessa testauksella halutaan arvioida ohjelmiston absoluuttista suorituskykyä vs. ohjelmiston suorituskykyä toteuttaa loppukäyttäjien liiketoimintavaatimukset. Kun nämä määrittelyt saadaan toteutettua hyvin, on suorituskykytestauksenkin kehittäminen tulevaisuudessa selkeämpää sekä siitä saavutettavat tulokset hyödyllisiä case-yrityksen kannalta.

7.3 Tutkimuksen laadun arviointi

Tutkimus toteutettiin hyödyntämällä suunnittelutieteellistä tutkimusmenetelmää (design science research) sekä kirjallisuuskatsausta. Suunnittelutieteellinen tutkimusmenetelmä menetelmä soveltui tutkimuksen tavoitteisiin hyvin, sillä tutkimusmenetelmän avulla pystyttiin kehittämään ja testaamaan uusia suorituskykytestauksen sekä -budjetoinnin menetelmiä tutkimuksen case-yrityksen tarpeisiin. Tutkimusmenetelmän avulla saavutettiin riittävästi konkreettista tietoa, mikä mahdollisti vastamaan tutkimuksessa asetettuihin tutkimuskysymyksiin. Tutkimuksessa toteutettiin yksi kappale iteraatioita suorituskykytestien osalta. Jotta tutkimuksessa olisi saatu vielä tarkempia tutkimustuloksia, olisi iteraatiokierroksia ollut hyvä suorittaa useampiakin. Toisaalta tutkimuksen keskittyessä vahvasti case-yrityksen tarpeisiin, olisivat mahdollisilla lisäiteraatioilla saavutetut mahdolliset tulokset palvelleet enemmän case-yrityksen tarpeita, kuin esimerkiksi uuden teorian tuottoa tutkimusaiheesta.

Kirjallisuuskatsauksen rooli tässä tutkimuksessa oli toimia tukevana sekä taustoittavana osana tutkimusmenetelmää. Kirjallisuuskatsauksen avulla taustoitettiin teoriapohjaa sekä viitekehystä suorituskykybudjetista, joiden avulla mahdollistettiin käytännön toteutukset suorituskyvyn määrittämisestä sekä mittareista. Koska suorituskykybudjetista on käsitteenä varsin vähän akateemisia lähteitä, koostui katsaus suurilta osin erilaisista dokumentaatioista sekä kirjoituksista, mitkä löytyivät alan eri julkaisuista. Tämä voidaan nähdä kritiikkinä kirjallisuuskatsausta kohtaan. Vaikka katsaus koostuukin pitkälti edellä mainituista lähteistä, käsittelevät ne aihetta keskenään hyvin samalla

tavalla. Lähteitä on myös pyritty saamaan aiheesta mahdollisimman paljon. Tällä tavoin voidaan perustella käytettyjen lähteiden relevanttiutta kirjallisuuskatsauksen rakentamisessa.

Tutkimuksen rajoituksena voidaan nähdä myös sen keskittymisen yksittäin case-yrityksen tuotekehitysympäristöön. Vaikka tulokset ovatkin case-yrityksen näkökulmasta hyödyllisiä, niiden yleistettävyys muihin ohjelmistoympäristöihin voi olla rajoittunutta. Tämän tutkimuksen tuloksia määrittävät pitkälti case-yrityksen eri kehitysympäristöt sekä käytössä olevat teknologiat. Suorituskykybudjetoinnin sekä -testauksen menetelmät vaativat kuitenkin hyvin todennäköisesti muutoksia erilaisissa ympäristöissä, joissa suorituskyvyn hallintaan vaikuttavat monet erilaiset tekijät. Siksi tämän tutkimuksen tuloksia ei välttämättä voida soveltaa muissa ympäristöissä suoraan, mutta tutkimuksessa löydettyjä menetelmiä sekä käsitteitä voidaan hyödyntää pohjana erilaisille suorituskyvyn budjetointi- sekä testauskokonaisuuksille.

7.4 Jatkotutkimusaiheet

Tutkimuksesta nousi esiin seuraavia jatkotutkimusaiheita, joilla olisi mahdollista parantaa suorituskykybudjetoinnin sekä -testauksen prosesseja. Suorituskykybudjetin osalta olisi hyvä tutkia sitä, mistä budjetointiprosessissa tulisi lähteä liikkeelle, mitä osa-alueita olisi tarpeen mitattavan ja millaiset työkalut olisivat parhaita budjetointiprosessi tueksi. Toinen tutkimusaihe suorituskykybudjetoinnissa voisi keskittyä tutkimaan, onko olemassa ns. ”oikeaa” tapaa suorituskykybudjetin määrittämiselle. Tulevissa tutkimuksessa voitaisiin selvittää, mitä kaikkia näkökulmia tulisi huomioida ja millaisia mittareita voitaisiin kehittää tarkemman suorituskykybudjetin saavuttamiseksi. Esimerkiksi räätälöityjen mittareiden kehittäminen eri ohjelmistojen tarpeisiin voisi parantaa suorituskykybudjetoinnin prosesseja. Tämän tutkimuksen tulokset osoittavat, että mittareiden ja raja-arvojen asettamine on kriittinen osa budjetointia, mutta nykyiset lähestymistavat ovat vielä melko yleisluontoisia. Jatkotutkimuksissa voitaisiin paneutua tarkemmin siihen, miten erilaisia suorituskyvyn osa-alueita voidaan mitata tarkemmin ja näin kohdistaa budjetointia entistä tehokkaammin.

Suorituskykytestauksen osalta olisi hyvä tutkia käytettyjen testausratkaisujen skaalautuvuutta sekä rajoituksia. Etenkin case-yritykselle valitussa nykyisessä ratkaisussa skaalautuvuus voi muodostua tulevaisuudessa ongelmaksi. Skaalautuvuuteen vaikuttavat sekä käytettävät ratkaisut että tapa, joilla testejä luodaan. Nykyisessä toteutuksessa käytettyjen ratkaisujen osalta voidaan saavuttaa piankin tilanne, jossa käytettyjen ratkaisujen resurssirajoitteet alkavat rajoittamaan suorituskykytestausta. Tällöin testit eivät enää kykene antamaan täyttä kuvaa suorituskyvyn nykytilasta, jolloin testien merkitys katoaa. Tämän lisäksi nykyinen tapa kirjoittaa suorituskykytestejä on työläs, sillä uusien testitapausten rakentaminen täytyy aina aloittaa nollasta, etenkin suurien ja eri käyttötapauksia edustavien lähtödatojen osalta. Tähän olisikin hyvä tutkia

ratkaisuja, joilla tällaisten testitapausten rakentaminen saataisiin sulavammaksi ja skaalautuvammaksi ilman, että testien kirjoittaminen hidastaa liikaa itse kehitystyötä.

8 YHTEENVETO

Tämän tutkielman tutkimusongelmana oli määrittää tutkimuksen case-yritykselle ohjelmiston suorituskyvyn testaamiseen käytettävät suorituskykytestit, sekä suorituskykybudjetti valvomaan ohjelmiston suorituskyvyn tilaa sekä kehitystä. Tutkielman tavoitteen oli toteuttaa konkreettisesti prosessi suorituskykytestaukselle, sekä määrittää suorituskykybudjetti osaksi luotavia testejä. Tämän määrittelyn pohjalta laadittiin seuraava päätutkimuskysymys, sekä tälle kaksi taustoittavaa tutkimuskysymystä.

- Miten suorituskykytestaus voidaan toteuttaa ja budjetoida
    - Miten suorituskykyä voidaan budjetoida?
    - Miten suorituskykytestaus voidaan toteuttaa?

Tutkielman rakenne muodostuu teoreettisesta sekä empiirisestä osiosta. Teoreettisen osuuden sisältö keskittyy määrittelemään ohjelmistotestauksen sekä suorituskykytestauksen keskeisiä käsitteitä. Teoreettisessa osuudessa toteutetaan myös kirjallisuuskatsaus, jonka tarkoituksena on taustoittaa toista tutkimuskysymystä. Teoreettisen osuuden pohjalta luodaan tutkielman teoreettinen viitekehys, jonka avulla tutkimuskysymyksiä lähdetään ratkaisemaan tutkielman empiirisessä osiossa.

Ensimmäiseen taustoittavaan tutkimuskysymykseen vastataan yleisellä tasolla jo tutkielman kirjallisuuskatsauksessa. Katsauksen avulla saavutettiin käsitys siitä, mitä suorituskykybudjetointi on, mitä se voi sisältää, ja kuinka budjetointiprosessia voidaan lähestyä. Kirjallisuuskatsauksessa nousi esiin, että suorituskyvyn budjetointi on hyvin riippuvainen siitä, mitä budjetoitavalla ohjelmistolla halutaan saavuttaa, ja minkälainen on ohjelmiston käyttöympäristö sekä -tarkoitus. Budjetointi vaatii myös selkeää näkemystä ohjelmiston tuotekehitystiimiltä siitä, mihin ohjelmistoa halutaan tulevaisuudessa viedä. Nämä päätökset ohjaavat budjetointiprosessia sekä budjetin sisältöä.

Toiseen tausoittavaan tutkimuskysymykseen vastattiin tutkielman teoriaosuudessa, jossa käsiteltiin ohjelmistotestauksen sekä suorituskykytestauksen käsitteitä. Osuudessa nousi molemmista käsitteistä esiin se, että onnistuneen testauksen takana on tunnistaa käyttötarkoitukseen oikeat

testauksen tasot, tyypit, sekä kohteet. Näinollen luotava testaus voidaan rakentaa palvelemaan parhaiten sille asetettuja testaustavoitteita.

Tutkielman tuloskappaleessa pyrittiin vastaamaan tutkielman päätutkimuskysymykseen. Kappaleessa hyödynnettiin suunnittelutieteellistä tutkimusmenetelmää, minkä avulla tuotetaan jonkin ratkaisu aiemmin määriteltyyn ongelmaan. Tämän jälkeen ratkaisusta saadun palautteen perusteella suoritetaan riittävä määrä iteraatioita, joilla ratkaisua pyritään kehittämään. Päätutkimuskysymykseen vastattiin luomalla tutkielmassa suorituskykybudjetti sekä suorituskykytestit valvomaan case-yrityksen ohjelmiston suorituskyvyn kehittymistä tulevaisuuden muutosten alla. Suorituskykytesteistä annettiin case-yritykselle muutama teknologiavaihtoehto annettujen määrittelyjen pohjalta, ja palautteen perusteella toteutettiin varsinainen suorituskykytestaus valitulla teknologialla. Valmis suorituskykytestaus sisältää tunnistettujen testauskohteiden testaamisen halutulla testidatalla, sekä suorituskykybudjetin raja-arvot eri testauskohteiden suorituskyvyn seurantaan.

Yhteenvetona tutkimus osoitti, että suorituskykytestaus ja -budjetointi ovat keskeisiä työkaluja ohjelmiston laadunhallinnassa, ja niiden tehokas toteuttaminen voi merkittävästi parantaa ohjelmiston suorituskykyä ja käyttäjäkokemusta. Tutkimuksen perusteella suorituskykybudjetointi on keino, jolla voidaan hallita ja optimoida suorituskyvyn eri osa-alueita, mutta sen käytännön toteutus vaatii tarkkaa tasapainottelua ja jatkokehittämistä. Suorituskyvyn tehokkaan testaamisen sekä seurannan pohjana voidaan nähdä kaksi osa-aluetta: ensimmäisenä on tärkeää tunnistaa riittävällä tasolla sekä testattava kohde, että testauksen tavoitteet. Näin voidaan varmistaa oikea lähestymistapa testauksen toteuttamiselle. Toinen osa-alue on tunnistaa oikeat ja ohjelmiston kontekstin kannalta tärkeimmät toiminnalliset tavoitteet. Tätä kautta on mahdollista toteuttaa suorituskykybudjetti, josta saatava palaute ohjaa testattavan ohjelmiston kehitystä sille haluttuun suuntaan.

LÄHTEET

About GitHub and Git. (n.d.). GitHub Docs. Retrieved April 20, 2024, from https://docs.github.com/_next/data/X9QLXRAvRjjbP_KZ-s8Md/en/free-pro-team@latest/get-started/start-your-journey/about-github-and-git.json?versionId=free-pro-team%40latest&productId=get-started&restPage=start-your-journey&restPage=about-github-and-git

Amannejad, Y., Gəruslu, V., Irving, R., & Sahaf, Z. (2014). A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study. In *Proceedings – IEEE 7th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2014* (p. 311). https://doi.org/10.1109/ICSTW.2014.34

Apache JMeter – User's Manual. (n.d.). Retrieved August 31, 2024, from https://jmeter.apache.org/usermanual/

Black, R. (2009). *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing* (Vol. 3). Wiley Publishing Inc.

Boukhlif, M., Hanine, M., & Kharmoum, N. (2023). A Decade of Intelligent Software Testing Research: A Bibliometric Analysis. *Electronics*, *12*(9), Article 9. https://doi.org/10.3390/electronics12092109

Everett, G. D., & McLeod, R. (2007). *Software Testing: Testing Across the Entire Software Development Life Cycle* (1st ed.). Wiley. https://doi.org/10.1002/9780470146354

Fewster, M., & Graham, D. (1994). *Software Test Automation*. ACM Press.

Fowler, M. (2018, January 16). *Integration Test*. Martinfowler.Com. https://martinfowler.com/bliki/IntegrationTest.html

GitHub Actions documentation. (n.d.). GitHub Docs. Retrieved April 20, 2024, from https://docs.github.com/en/actions

Hernández, C. M., Martínez, A., Quesada-López, C., & Jenkins, M. (2021). Comparison of End-to-End Testing Tools for Microservices: A Case Study. In Á. Rocha, C. Ferrás, P. C. López-López, & T. Guarda (Eds.), *Information Technology and Systems* (pp. 407–416). Springer International Publishing. https://doi.org/10.1007/978-3-030-68285-9_39

Hourani, H., Hammad, A., & Lafi, M. (2019). The Impact of Artificial Intelligence on Software Testing. *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, 565–570. https://doi.org/10.1109/JEEIT.2019.8717439

Huizinga, D., & Kolawa, A. (2007). *Automated Defect Prevention: Best Practices in Software Management*. John Wiley & Sons, Inc., Hoboken, New Jersey.

ISTQB. (2023, April 21). *Certified Tester. Foundation Level Syllabus. V4.0*. https://www.istqb.org/certifications/certified-tester-foundation-level

Jormakka, R., Koivusalo, K., Lappalainen, J., & Niskanen, M. (2015). *Laskentatoimi* (Vol. 4). Edita Publishing Oy.

Kangasniemi, M., Utriainen, K., Ahonen, S.-M., Pietilä, A.-M., Jääskeläinen, P., & Liikanen, E. (2013). Kuvaileva kirjallisuuskatsaus: Eteneminen tutkimuskysymyksestä jäsennettyyn tietoon. *Hoitotiede*, *25*.

Lemonsoft Oyj – Kotimainen yritysohjelmisto liiiketoiminnallesi. (n.d.). Lemonsoft. Retrieved October 13, 2024, from https://www.lemonsoft.fi/lemonsoft-oy/

Lukka, K. (2014, May 19). Kari Lukka: Konstruktiivinen tutkimusote. *METODIX*. https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/

Luo, Q., Nair, A., Grechanik, M., & Poshyvanyk, D. (2017). FOREPOST: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering*, *22*(1), 6–56. https://doi.org/10.1007/s10664-015-9413-5

Medium What is a Performance Budget and how to set a Performance Budget for your site? (2020, November 30). *W3 SpeedUp*. https://medium.com/w3-speedup/what-is-a-performance-budget-and-how-to-set-a-performance-budget-for-your-site-d87cda331e93

Melkozerova, O. M., & Rassomakhin, S. G. (2020). Software performance testing. *Bulletin of V.N. Karazin Kharkiv National University, series «Mathematical modeling. Information technology. Automated control systems»*, *45*, 56–66. https://doi.org/10.26565/2304-6201-2020-45-07

Mitä on suorituskykytestaaminen? Tyypit, käytännöt, työkalut ja paljon muuta! (n.d.). Retrieved March 9, 2024, from https://www.zaptest.com/fi/mita-on-suorituskykytestaaminen-syvasukellus-tyyppeihin-kaytantoihin-tyokaluihin-haasteisiin-ja-muuhun

Neilson, J. E., Woodside, C. M., Petriu, D. C., & Majumdar, S. (1995). Software bottlenecking in client-server systems and rendezvous networks. *IEEE*

Transactions on Software Engineering, *21*(9), 776–782. IEEE Transactions on Software Engineering. https://doi.org/10.1109/32.464543

Neves, L., Campos, O., Santos, R., C. de Magalhaes, C., Santos, I., & de Souza Santos, R. (2024). *Elevating Software Quality in Agile Environments: The Role of Testing Professionals in Unit Testing*.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Performance budgets 101 | Articles | web.dev. (n.d.). Retrieved April 12, 2024, from https://web.dev/articles/performance-budgets-101

Playwright. (n.d.). Retrieved August 31, 2024, from https://playwright.dev/

Rodrigues, A. G., Demion, B., & Mouawad, P. (2019). *Master Apache JMeter – From Load Testing to DevOps: Master Performance Testing with JMeter*. Packt Publishing. https://viewer-ebscohost-com.ezproxy.jyu.fi/EbscoViewerService/ebook?an=2223296&callbackUrl=https%3a%2f%2fresearch.ebsco.com&db=e000xww&format=EB&profId=ehost&lpid=&ppid=&lang=en&location=https%3a%2f%2fresearch-ebsco-com.ezproxy.jyu.fi%2fc%2fx3kxfd%2fsearch%2fdetails%2f5a2vgnhkmn%3fdb%3de000xww&isPLink=False&requestContext=&profileIdentifier=x3kxfd&recordId=5a2vgnhkmn

Salminen, A. (2011). *Mikä kirjallisuuskatsaus? : Johdatus kirjallisuuskatsauksen tyyppeihin ja hallintotieteellisiin sovelluksiin – Osuva*. https://osuva.uwasa.fi/handle/10024/7961

School of Information Technology, Vellore Institute of Technology, Vellore (Tamil Nadu), India., Chevuturu, A. A., Mathur, D. P., School of Information Technology, Vellore Institute of Technology, Vellore (Tamil Nadu), India., Reddy, B. J. P. K., School of Information Technology, Vellore Institute of Technology, Vellore (Tamil Nadu), India., R, Dr. C., & Assistant Professor, School of Information Technology, Vellore Institute of Technology, Vellore (Tamil Nadu), India. (2022). A Comparative Survey on Software Testing Tools. *International Journal of Engineering and Advanced Technology*, *11*(6), 32–40. https://doi.org/10.35940/ijeat.F3664.0811622

Siddiqui, K. H., & Woodside, C. M. (2002). Performance aware software development (PASD) using resource demand budgets. *Proceedings of the 3rd International Workshop on Software and Performance*, 275–285. https://doi.org/10.1145/584369.584412

S.M.K, Q., & Farooq, S. U. (2010). Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications*, *6*. https://doi.org/10.5120/1343-1448

Sommerville, I. (2011). *Software Engineering* (Vol. 9). Pearson Education Inc.

TEEMME LIIKETOIMINNASTA ENNUSTETTAVAA - Finazilla. (n.d.). Retrieved March 9, 2024, from https://www.finazilla.fi/

The International Organization for Standardization (ISO) & The International Electrotechnical Comisison (IEC). (n.d.). *ISO/IEC 25010:2011(en), Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Retrieved March 21, 2024, from https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

The Risks of Performing No Testing or Minimal Testing. (n.d.). Retrieved May 8, 2024, from https://muuktest.com/blog/not-testing-software

Web performance | MDN. (2023, December 5). https://developer.mozilla.org/en-US/docs/Web/Performance/Performance_budgets

What is Azure Load Testing? (2023, May 11). https://learn.microsoft.com/en-us/azure/load-testing/overview-what-is-azure-load-testing

What is Azure – Microsoft Cloud Services | Microsoft Azure. (n.d.). Retrieved October 10, 2024, from https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure

What Is Performance Budget? | Crystallize. (n.d.). Retrieved April 12, 2024, from https://crystallize.com/answers/tech-dev/what-is-performance-budget