SAVONIA
ammattikorkeakoulu

OPINNÄYTETYÖ - AMMATTIKORKEAKOULUTUTKINTO
TEKNIIKAN JA LIIKENTEEN ALA

TESTIAUTOMAATION TEHOSTAMINEN TESTITAPAUSTEN PRIORISOINNIN AVULLA

TEKIJÄT Niilo Viljamaa

Riku Malm

SAVONIA-AMMATTIKORKEAKOULU	OPINNÄYTETYÖ Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Tutkinto-ohjelma Tietotekniikan tutkinto-ohjelma	
Työn tekijät Niilo Viljamaa, Riku Malm	
Työn nimi Testiautomaation tehostaminen testitapausten priorisoinnin avulla	
Päiväys 30.5.2024	Sivumäärä/Liitteet 45
Toimeksiantaja/Yhteistyökumppani(t) Ponsse Oyj	

Tiivistelmä

Ohjelmistotestaus on olennainen osa ohjelmistokehitysprosessia, ja tehokkaat testiautomaatiotyökalut ovat välttämättömiä testausprosessin tehostamiseksi ja skaalaamiseksi. Opinnäytetyön tavoitteena oli kehittää palvelin, joka jakaa testisarjoja testiautomaatioympäristöjen kesken priorisointijärjestelmän avulla. Tämä ratkaisu pyrki parantamaan testausprosessin tehokkuutta ja vähentämään manuaalisen työn määrää.

Työ toteutettiin keväällä 2024 osaksi Ponsse Oyj:n Opti 5G -tietojärjestelmän testiautomaatioprosessia. Työn teoriaosassa käsiteltiin ohjelmistotestauksen merkitystä, testiautomaation hyötyjä, jatkuvan integraation ja jatkuvan toimituksen prosesseja sekä kuormanjaon teoriaa ja algoritmeja. Palvelin toteutettiin C#-ohjelmointikielellä hyödyntäen ASP.NET -ohjelmistokehystä. Priorisointijärjestelmän vaatimat tiedot aiemmista testisuorituksista tallennettiin SQLite-tietokantaan.

Opinnäytetyön aikana kehitetty palvelin saatiin valmiiksi koekäyttöä varten. Paikallisten testien perusteella todettiin, että palvelin jakaa testejä haluttujen prioriteettien mukaisesti ja se osaa huomioida myös ajoympäristöjen erikoisominaisuudet. Työ aiotaan seuraavaksi siirtää tuotantokäyttöön, jossa sen toimivuutta ja luotettavuutta voidaan arvioida todellisissa olosuhteissa.

Avainsanat

Testiautomaatio, ohjelmistotestaus, ASP.NET, kuormanjako

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study	
Technology, Communication and Transport	
Degree Programme Degree Programme in Information Technology	
Authors Niilo Viljamaa, Riku Malm	
Title of Thesis Enhancing test automation through test case prioritization	
Date 30.5.2024	Pages/Appendices 45
Client Organisation / Partners Ponsse Plc	

Abstract

Software testing plays a crucial role in the software development process and effective test automation tools are essential for streamlining and scaling the testing process. The aim of this thesis was to develop a server that distributes test suites among test automation environments using a prioritization system. This solution aimed to improve the efficiency of the testing process and reduce the amount of manual work involved.

The work was carried out in the spring of 2024 as part of the test automation process for the Ponsse Plc Opti 5G information system. The theoretical part of the work dealt with the importance of software testing, the benefits of test automation, continuous integration and continuous delivery processes, and load balancing theory and algorithms. The server was implemented in the C# programming language using the ASP.NET framework. The data required by the prioritization system from previous test runs was stored in an SQLite database.

The server developed during the thesis work was completed for pilot use. Local tests showed that the server distributes tests according to the desired priorities and is also able to take into account the special characteristics of the runtime environments. The work will then be transferred into production use, where its functionality and reliability can be evaluated under real-world conditions.

Keywords

Test automation, software testing, ASP.NET, load balancing

SISÄLTÖ

1 JOHDANTO 7
2 OHJELMISTOTESTAUS 9
2.1 Johdanto 9
2.2 Ohjelmistotestauksen teoria 9
2.3 Ohjelmistotestauksen kustannustehokkuus 10
2.4 Ohjelmistotestauksen kehitysmenetelmät 12
2.5 Testausmenetelmät 13
2.5.1 Mustalaatikkotestaus 14
2.5.2 Lasilaatikkotestaus 14
2.6 Funktionaalinen testaus 14
2.6.1 Yksikkötestaus 15
2.6.2 Integraatiotestaus 16
2.7 Järjestelmätestaus 17
2.8 Manuaalitestaus 17
2.9 Automaattitestaus 18
3 TESTIAUTOMAATIO 19
3.1 Hyödyt 19
3.2 Haasteet 20
3.3 Työkalut ja tekniikat 21
3.3.1 Testaustyökalut 21
3.3.2 Ohjelmointikielet 22
4 CICD JA DEVOPS 24
4.1 Johdanto 24
4.2 DevOps periaatteet 25
4.2.1 Työkalut ja teknologiat 26
4.3 Jatkuva integraatio ja jatkuva toimitus 26
4.3.1 Hyödyt 27
5 KUORMANJAON TEORIA JA ALGORITMIT 28
5.1 Johdanto 28
5.2 Staattinen kuormanjako 28
5.2.1 Kiertovuorottelu 29

5.2.2 Painotettu kiertovuorottelu 29

5.3 Dynaaminen kuormanjako 30

5.3.1 Hajautettu dynaaminen kuormanjako 31

5.3.2 Ei-keskitetty dynaaminen kuormanjako 31

6 TYÖN SUUNNITTELU 32

6.1 Vaatimusmäärittely 32

6.2 Tekninen suunnittelu 33

7 TOTEUTUS 35

7.1 Tietokanta 35

7.1.1 Jaettavien testien ylläpito 35

7.1.2 Testihistorian ylläpito 36

7.2 Palvelimen toiminta 36

7.2.1 Arkkitehtuurin yleiskuvaus 37

7.3 Testien jako 39

7.4 Testien priorisointi 40

7.5 Kuormanjako palvelimen näkökulmasta 40

8 POHDINTA JA JOHTOPÄÄTÖKSET 42

Kuva 1 Opti 5G:n työskentelynäkymä. (Ponsse Oyj, julkaisuaika tuntematon) 7

Kuva 2 Testauskustannusten ja tuoton välinen suhde. (Bierig;Brown;Galván;& Timoney, 2022, s. 4) 11

Kuva 3 Virheiden suhteellinen kustannus ohjelmiston elinkaaren aikana. (Dawson;Burrell;Rahim;& Stephen, 2010) 12

Kuva 4 Perinteinen vesiputousmalli ohjelmistotuotannossa (Patton, 2001, s. 33). 13

Kuva 5 Anichen versio testauspyramidista (Aniche, 2022, s. 23). 14

Kuva 6 Yksikkötesti painottuu yhteen yksikköön, kuten yksittäiseen luokkaan (Aniche, 2022, s. 20) 15

Kuva 7 Integraatiotestien avulla varmistetaan, että eri komponentit ovat yhteensopivia ja toimivat yhdessä virheettömästi. (Aniche, 2022, s. 21) 16

Kuva 8. Testiautomaation hyödyt (Jose, 2021, s. 11) 20

Kuva 9. Testiautomaation haasteet (Viljamaa, 2024) 21

Kuva 10. Jatkuvaa integraatiota (CI) ja jatkuvaa toimitusta (CD) kuvataan usein syklinä. 27

Kuva 11 Kuormanjakaja tasaa tulevat pyynnöt kolmen palvelimen kesken kiertovuorottelumenetelmällä. (Padilha, 2018, s. 6) 29

Kuva 12 Esimerkki painotetusta kiertovuorottelualgoritmista (Padilha, 2018, s. 7). 30

Kuva 13. Palvelin osana testiautomaatioprosessia (Viljamaa, 2024) 34

Kuva 14 Testisarjat-taulu tietokannassa. 35
Kuva 15 Aiempien testiajojen taulu tietokannassa. 36
Kuva 16 Yksinkertaistettu versio palvelimen arkkitehtuurista. 37
Kuva 17 Testit palauttava reitti. 38
Kuva 18 Esimerkki ympäristön lähettämästä konfiguraatiosta. 38
Kuva 19 Testisarjojen jakologiikka kaaviona. 39
Kuva 20 Testisarjat toimivat jaettavana kuormana. 41

1 JOHDANTO

Opinnäytetyön toimeksiantaja on suomalainen pörssiyritys Ponsse Oyj. Ponsse valmistaa, myy ja huoltaa metsäkoneita sekä niihin liittyviä tietojärjestelmiä. Yritys on tunnettu korkealaatuisista metsäkoneistaan, jotka on suunniteltu erityisesti kestävyyttä ja tehokkuutta silmällä pitäen vaativissa metsäolosuhteissa.

Opti 5G on Ponssen kehittämä tietojärjestelmä korjuutyön hallintaan. Arkipäivisin Opti 5G:stä julkaistaan uusi kehitysversio, jonka julkaisuprosessiin kuuluu myös testiautomaation testien suorittaminen. Testiautomaatio suoritetaan Robot Frameworkin avulla, ja se sisältää ohjelmiston testausta oikealla fyysisellä laitteistolla sekä täysin ohjelmistopuolella, jossa vaaditut fyysiset komponentit korvataan simuloiduilla versioilla. Testiympäristöjä on siis kahta eri tyyppiä:

1. Fyysinen laitteisto (HIL - Hardware in the loop)
   Testit suoritetaan oikealla laitteistolla mahdollisimman realististen käyttöolosuhteiden simuloimiseksi. Tämä on erityisen tärkeää turvallisuusominaisuuksien testaamisessa, joissa laitteiston ominaisuudet ja rajoitukset voivat vaikuttaa merkittävästi ohjelmiston toimintaan.
2. Simulointiympäristö (SIL – Software in the loop)
   Testit suoritetaan normaalisti, mutta fyysiset laitteet on korvattu ohjelmistotason simuloinnilla. Näissä ympäristöissä voidaan suorittaa pääasiassa samoja testejä, kuin fyysisissä ympäristöissäkin, muutamaa poikkeusta lukuun ottamatta.

Kuva 1 Opti 5G:n työskentelynäkymä. (Ponsse Oyj, julkaisuaika tuntematon)

Testiautomaatio koostuu laajasta valikoimasta testejä, joiden avulla varmistetaan tietojärjestelmän toimivuus ja luotettavuus. Nämä testit tarkkailevat esimerkiksi laitteiston virtamääriä ja varmistavat, että käyttöliittymä reagoi halutulla tavalla testien aikana. (Kuva 1)

Nykytilanteessa ajettavat testit määritetään ympäristölle Robot Frameworkin tagien avulla. Robot Framework on avoimen lähdekoodin testiautomaatiokehys, jota käytetään osana Ponssen ohjelmistotestausta. Tagien avulla voidaan ryhmitellä ja suodattaa testejä, mikä mahdollistaa tiettyjen testikokonaisuuksien ajamisen tietyissä ympäristöissä. Testimäärän kasvaessa tämä lähestymistapa kuitenkin on osoittautunut puutteelliseksi, sillä tällä ryhmittelyllä ei päästä vaikuttamaan tarpeeksi hienojakoisesti ajettaviin testeihin ilman kohtuuttoman suurta manuaalista työpanosta. Esimerkiksi uusien testien lisääminen voi olla haasteellista, sillä ne on sovitettava jo olemassa oleviin ympäristöjen testiajoihin niiden keston tasapainottamiseksi. Tämä voi vaatia testien siirtämistä ympäristöstä toiseen, mikä on aikaa vievää ja virhealtista. Testien määrän kasvaessa aikataulupaineet voivat johtaa siihen, että osa testeistä jätetään kokonaan ajamatta, mikä heikentää testauksen kattavuutta ja luotettavuutta.

Näihin haasteisiin vastaamiseksi kehitettiin palvelin, joka jakaa testejä eri ympäristöjen kesken automaattisesti ja tehokkaasti. Tämän ratkaisun avulla voidaan minimoida manuaalisen työn määrä ja varmistaa, että testit ajetaan optimaalisesti käytettävissä olevien resurssien puitteissa. Ajoympäristöjen lisääminen ei ollut vaihtoehto, joten palvelinratkaisu tarjoaa joustavan ja skaalautuvan tavan hallita kasvavia testimääriä. Palvelinratkaisu hyödyntää hienojakoisempaa priorisointijärjestelmää, joka perustuu useisiin tekijöihin, kuten testien tärkeyteen ja aiemmissa testiajoissa ilmenneiden virheiden määrään.

2 OHJELMISTOTESTAUS

2.1 Johdanto

1960- ja 70-lukujen taitteessa ohjelmistoalaa leimasi merkittävä käänne, jota kutsuttiin ohjelmistokriisiksi (software crisis). Ohjelmistoala kehittyi nopeasti ja kysyntä uusille ohjelmistoille kasvoi räjähdysmäisesti. Tietokoneiden kehittyessä myös ohjelmistot kävivät yhä monimutkaisemmiksi, mikä vaikeutti niiden kehitystä ja ylläpitoa. Tämä johti siihen, että ohjelmistoja kehitettiin kiireellä ilman asianmukaista suunnittelua ja testausta. (Bierig;Brown;Galván;& Timoney, 2022, s. 1)

Kriisi toi mukanaan merkittäviä haasteita ohjelmistoalalle. Budjetit ja aikataulut paisuivat hallitsemattomasti, johtaen projektien viivästymiseen ja kustannusten nousuun. Kehitetyt ohjelmistot kärsivät lukuisista virheistä ja puutteista, heikentäen niiden käyttökelpoisuutta ja luotettavuutta, eivätkä ohjelmistot usein vastanneet niille asetettuja vaatimuksia. Ohjelmistojen kehitys ja ylläpito kävivät yhä hankalammiksi, mikä nosti kustannuksia ja teki uusien ominaisuuksien ja korjausten käyttöönotosta työläämpää. (Bierig;Brown;Galván;& Timoney, 2022, s. 2)

Nämä haasteet pakottivat ohjelmistoalan etsimään uusia ratkaisuja ja menetelmiä ohjelmistokehityksen parantamiseksi. Tämä oli käännekohta, joka johti modernin ohjelmistotekniikan kehittymiseen ja ohjelmistotestauksen vakiintumiseen tärkeänä osana nykyaikaista sovelluskehitystä. Testiautomaatiotyökalujen ja -kirjastojen käyttöönotto ja kehitys ovat olleet ratkaisevassa asemassa ohjelmistotestauksen tehostamisessa ja ohjelmistokehityksen laadunvarmistuksessa. (Bierig;Brown;Galván;& Timoney, 2022, s. 3)

Laadukas ohjelmistotestaus on olennainen osa nykyaikaista ja tehokasta ohjelmistokehitystä. Sen avulla vähennetään ohjelmistossa olevien virheiden määrää, parannetaan sen luotettavuutta ja käyttäjäkokemusta. Ohjelmistotestaus on jatkuva prosessi, jota tulisi suorittaa koko ohjelmiston elinkaaren ajan, jotta varmistetaan sen toimivuus ja laatu.

2.2 Ohjelmistotestauksen teoria

Ohjelmistotestaus on prosessi, joka varmistaa, että ohjelmistoratkaisu tai -tuote täyttää sille asetut vaatimukset ja odotukset. Tavoitteena on myös havaita virheitä ja todentaa, että ohjelmisto sopii hyvin tarkoitukseensa. Toiminnallisten ja ei-toiminnallisten vaatimusten varmistamiseen käytetään monia erilaisia testausmenetelmiä ja -tekniikoita. (Jose, 2021, s. 3)

Ohjelmistotestauksen teorian tavoitteena on pystyä tunnistamaan ideaaliset testit – eli pienin mahdollinen määrä testidataa, joka tarvitaan varmistamaan, että ohjelmisto toimii odotetusti kaikilla syötteillä (Bierig;Brown;Galván;& Timoney, 2022, s. 11).

Goodenoughin ja Gerhartin mukaan (1975, ss. 492–508) testidatan tulisi olla

1. Luotettavaa: Testisyötteiden tulisi johdonmukaisesti tuottaa onnistuneita testituloksia, tai epäonnistuneita testituloksia. Luotettava testidata varmistaa, että testit tuottavat merkityksellisiä tuloksia ja niiden pohjalta voidaan arvioida luotettavasti ohjelman toimintaa erilaisissa tilanteissa. Syötteiden tulisi siis heijastaa todellisia käyttötapauksia mahdollisimman tarkasti.

2. Pätevää: Pätevän testidatan avulla pystytään havaitsemaan ohjelman mahdolliset virheet ja puutteet tehokkaasti. Sen tulisi kattaa mahdollisimman laaja valikoima erilaisia syötteitä ja tilanteita, mukaan lukien tyypillisiä käyttötapauksia, odotettuja syötteitä ja normaaleja toimintaympäristöjä, sekä ääritilanteita, kuten virheellisiä syötteitä, poikkeustilanteita ja epätavallisia käyttötapauksia.
3. Riittävää: syötteiden määrän tulisi olla riittävän suuri, jotta sen avulla kaikki ohjelmiston virheet ovat löydettävissä. Teoriassa tämä tarkoittaa sitä, että testidatan tulisi kattaa kaikki mahdolliset syötteet ja tilanteet, joita ohjelma voi kohdata.

Jos testiajo suoritetaan luotettavilla ja pätevillä testisyötteillä, voidaan olla kohtuullisen varmoja siitä, että ohjelma toimii odotetulla tavalla eri tilanteissa. Kuitenkin testidatan riittävyyttä arvioitaessa törmätään haasteeseen: kaikkien mahdollisten syötteiden testaaminen on käytännössä mahdotonta lukuun ottamatta vain kaikkein yksinkertaisimpia ohjelmia. Tätä lähestymistapaa, jossa pyritään testaamaan kaikki mahdolliset syötteet, kutsutaan kokonaisvaltaiseksi tai tyhjentäväksi testaukseksi (Bierig;Brown;Galván;& Timoney, 2022, s. 11). Oletetaan, että haluamme testata ohjelmaa, joka laskee yhteen kaksi 64-bittistä lukua. Kokonaisvaltainen testaus tässä tapauksessa tarkoittaisi kaikkien mahdollisten $2^{64} * 2^{64} = 2^{128}$ lukuparien testaamista.

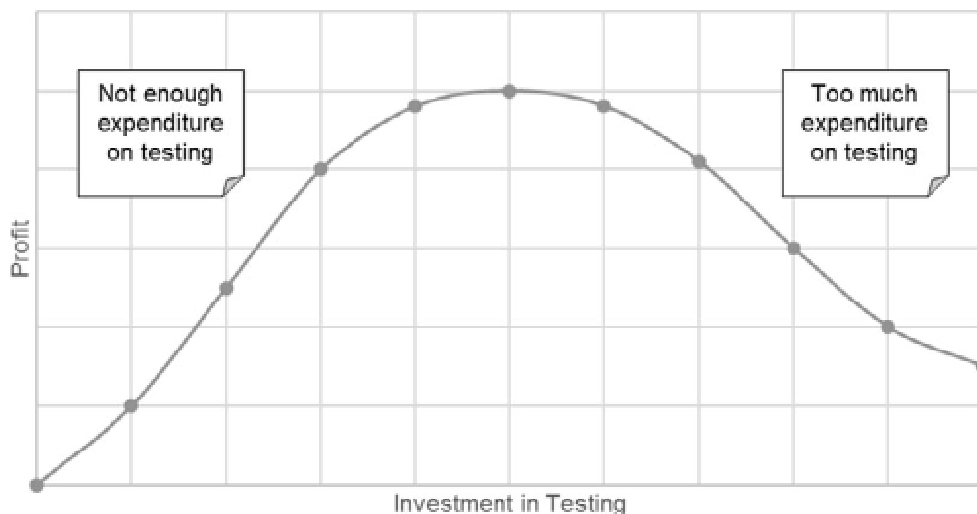
Vaikka ihanteellisessa tilanteessa ohjelmat voitaisiin testata kokonaisvaltaisesti, tämä ei ole käytännössä toteutettavissa lähes koskaan, sillä se vaatisi valtavan määrän aikaa ja resursseja. Tämän vuoksi on löydettävä tapoja vähentää testisyötteiden määrää ilman, että se heikentää testauksen laatua ja tehokkuutta. Ajankäytön ja resurssien optimoimiseksi testien tulisi löytää virheet suurella todennäköisyydellä, eli testisyötteiden ja -tapausten tulisi kohdistua niihin ohjelmiston osiin, joissa virheet ovat todennäköisimpiä. Lisäksi testitapausten päällekkäisyyksiä tulisi välttää, jotta varmistetaan testauksen tehokkuus ja kattavuus. Jos testit ovat päällekkäisiä, sama ohjelmiston toiminto tai osa-alue testataan useilla eri testeillä, mikä tuhlaa aikaa ja resursseja. Jokaisella testitapauksella tulisi olla selkeä tavoite ja se tulisi suunnitella testaamaan tiettyä toimintoa tai osa-aluetta (Bierig;Brown;Galván;& Timoney, 2022, ss. 11-14).

2.3 Ohjelmistotestauksen kustannustehokkuus

Tunnettu tietojenkäsittelytieteilijä Dijkstra (1970, s. 7) kirjoitti: Ohjelmistotestauksella voidaan osoittaa virheiden olemassaolo, mutta ei koskaan niiden puuttumista. Lauseesta on sittemmin tullut laajasti tunnettu ja sitä pidetään yhtenä ohjelmistotestauksen perusperiaatteista. Vaikka laajempi testaaminen voi löytää enemmän virheitä, mikään testisarja ei pysty varmistamaan ohjelmiston täydellistä virheettömyyttä. Testit voivat ainoastaan varmistaa, että testatut ominaisuudet toimivat odotetulla tavalla.

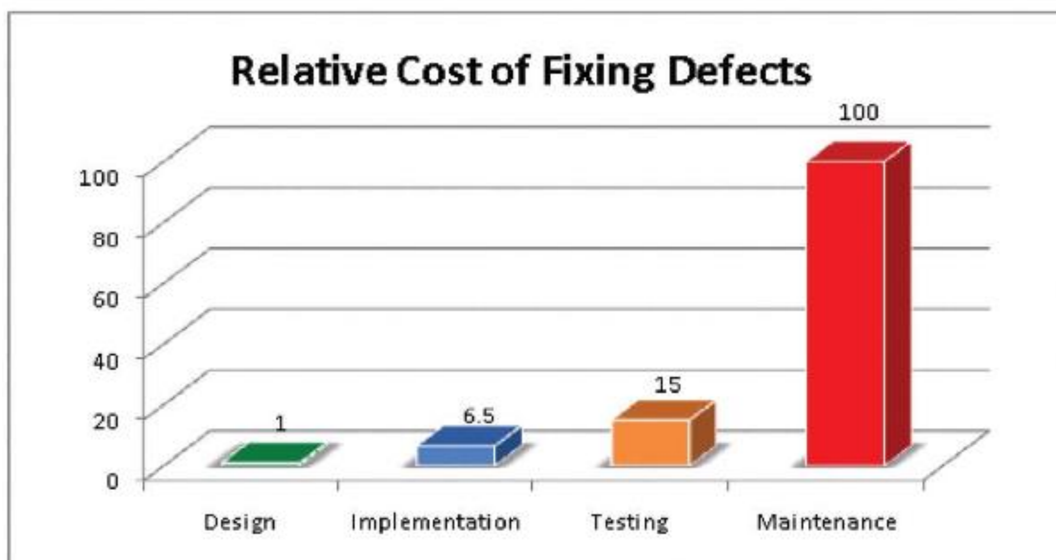
Ohjelmistotestausta voidaan pitää osana riskienhallintaa. Yritykselle ohjelmistojen virheillä voi olla sekä välittömiä että pitkän aikavälin seurauksia. Lyhyellä aikavälillä kustannukset liittyvät ensisijaisesti ohjelman korjaamiseen, joka saattaa viivästyttää sen julkaisua johtaen tulonmenetyksiin. Pitkällä aikavälillä viallisen ohjelmiston toimittamisesta yritykselle koituva mainehaitta voi laskea myyntiä ja vaikeuttaa uusien asiakkaiden hankintaa. (Aniche, 2022, ss. 14-15.)

Testauskustannukset tulisi suhteuttaa sekä yrityksen tuloihin että ohjelmistovioista johtuviin mahdollisiin kustannuksiin. Testausta voi tehostaa aloittamalla ohjelmiston testaaminen ja testien tekeminen jo kehityksen alkuvaiheessa. Ohjelmistotestausta voidaan pitää optimointiprosessina, jossa pyritään saamaan paras mahdollinen tuotto sijoitukselle. Investointi testaukseen vähentää ohjelmistovirheiden kustannuksia, mutta lisää kehityksen kustannuksia. Tärkeintä on löytää sopiva tasapaino näiden kustannusten välille. (Bierig;Brown;Galván;& Timoney, 2022, s. 4)



Kuva 2 Testauskustannusten ja tuoton välinen suhde. (Bierig;Brown;Galván;& Timoney, 2022, s. 4)

Kuvaaja havainnollistaa testaukseen käytettävien resurssien ja odotetun tuoton välistä suhdetta. Lopulta saavutetaan piste, jossa lisäinvestointi testaukseen ei enää johda voiton kasvuun, vaan sen sijaan se vain kasvattaa kustannuksia ilman havaittavaa hyötyä. Tämä on optimaalinen tasapaino testaukseen käytettävien resurssien ja siitä saatavan hyödyn välillä. Vaikka kuvaaja toimii hyödyllisenä pohjana investoinnin ja voiton tasapainon löytämiselle, on tärkeää huomata, että se on yksinkertaistettu malli. Todellisuudessa täydellinen tasapaino vaihtelee tapauskohtaisesti ja riippuu useista tekijöistä, kuten ohjelmiston tyypistä ja liiketoiminnan tavoitteista. (Kuva 2)

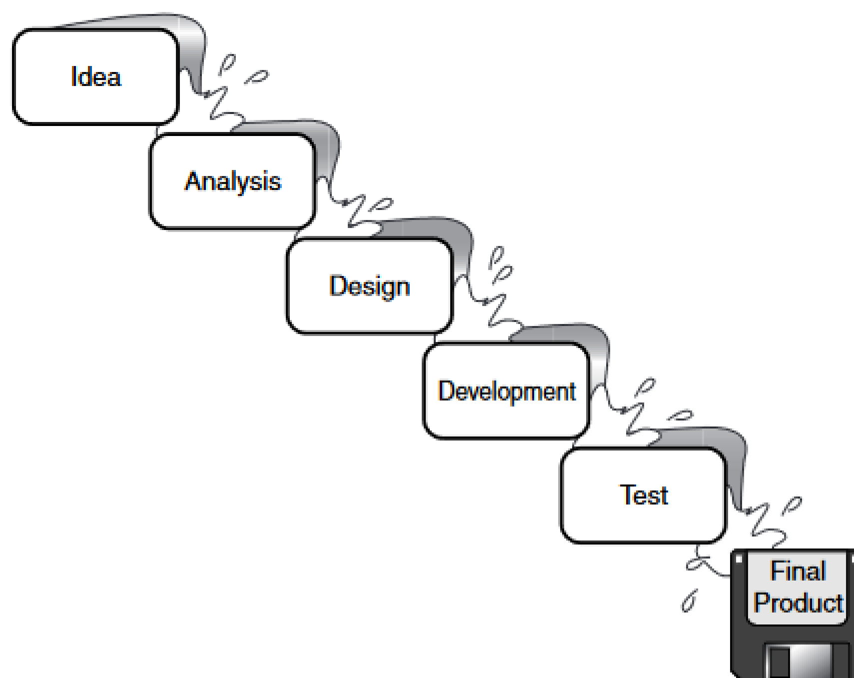


Kuva 3 Virheiden suhteellinen kustannus ohjelmiston elinkaaren aikana. (Dawson;Burrell;Rahim;& Stephen, 2010)

Vaikka testauksen aloittamiseen liittyvät kustannukset voivat olla korkeat, sen avulla voidaan välttää suurempia kustannuksia ja ongelmia myöhemmin. Aikainen testaaminen on huomattavasti halvempaa kuin virheiden korjaaminen myöhemmissä vaiheissa (Boehm & Papaccio, 1988). Lisäksi laadukas testaus voi auttaa parantamaan ohjelmiston laatua ja luotettavuutta, mikä voi vähentää kustannuksia ja parantaa asiakastyytyväisyyttä. (Kuva 3)

2.4 Ohjelmistotestauksen kehitysmenetelmät

Vesiputousmalli on perinteinen lähestymistapa ohjelmiston kehitykseen, jossa työ etenee vaiheittain ja lineaarisesti. Jokainen vaihe on suoritettava ennen seuraavan aloittamista. Vesiputousmallin keskeinen idea on käyttää runsaasti aikaa alussa projektin vaatimusmäärittelyyn ja suunnitteluun. Näin ollen aikaa ei kulu virheellisten vaatimusten perusteella tehdyn suunnittelun tai virheellisen suunnittelun pohjalta kirjoitetun koodin luomiseen. (Bierig;Brown;Galván;& Timoney, 2022, s. 280)



Kuva 4 Perinteinen vesiputousmalli ohjelmistotuotannossa (Patton, 2001, s. 33).

Vaikka vesiputousmalli on helposti ymmärrettävä ja selkeä, se kärsii jäykkyydestä ja joustavuuden puutteesta. Virheet havaitaan usein vasta myöhäisissä vaiheissa, mikä tekee niiden korjaamisesta kalliimpaa ja vaikeampaa. Ohjelmiston julkaisu myös viivästyy, kunnes kaikki löydetyt virheet ja ongelmat on saatu korjattua. Vesiputousmallia käytetään vieläkin joissakin projekteissa, mutta se on menettänyt suosiotaan ketterien kehitysmenetelmien, kuten Scrum ja Kanban, yleistyessä. (Kuva 4)

Koska perinteisissä ohjelmistokehitysmalleissa testaus suoritetaan vasta viimeisenä vaiheena ennen tuotteen julkaisemista, virheiden löytäminen ja korjaaminen on kallista ja aikaa vievää. Tämän seurauksena lopputuote saattaa sisältää virheitä, jotka voivat vaikuttaa sen vakauteen ja käyttäjäkokemukseen. Shift left -käytäntö pyrkii ratkaisemaan tätä ongelmaa siirtämällä testaustoimintoja mahdollisimman aikaiseen vaiheeseen kehitysprosessia. (Magowan, julkaisuaika tuntematon)

Vesiputousmallin sijaan nykyään suositaan ketteriä kehitysmenetelmiä, joista suosituin on Scrum. Sen sijaan, että testaus olisi oma erillinen vaiheensa, kuten vesiputousmallissa, se tapahtuu jatkuvasti ja samanaikaisesti kehityksen kanssa. Ketterissä menetelmissä tiimi tekee pieniä, toimivia osia tuotteesta lyhyiden kehitysjaksojen aikana (sprintit), ja jokaisen sprintin lopussa uutta toiminnallisuutta testataan välittömästi. Tämä mahdollistaa jatkuvan palautteen keräämisen ja nopean sopeutumisen muutoksiin. (Son, 2023)

2.5 Testausmenetelmät

Ohjelmistojen testaukseen on useita menetelmiä ja niitä voidaan luokitella eri tavoin. Yksi yleinen luokittelutapa perustuu testaajan tietämykseen ohjelmiston sisäisestä rakenteesta. Tämän lähestymistavan mukaan testausmenetelmät jaetaan yleisesti kahteen pääluokkaan: mustalaatikkotestaukseen (engl. black box testing) ja lasilaatikkotestaukseen (engl. white box testing). (Patton, 2001, ss. 55-56)

2.5.1 Mustalaatikkotestaus

Mustalaatikkotestaus on ohjelmistojen testausmenetelmä, jossa keskitytään ohjelman ulkoiseen toimintaan ilman tietämystä sen sisäisestä toteutuksesta tai lähdekoodista. Mustalaatikkotestaus perustuu ohjelman spesifikaatioon, ja sen tavoitteena on varmistaa, että ohjelma toimii odotetulla tavalla ja täyttää sille annetut vaatimukset. (Bierig;Brown;Galván;& Timoney, 2022, s. 142)

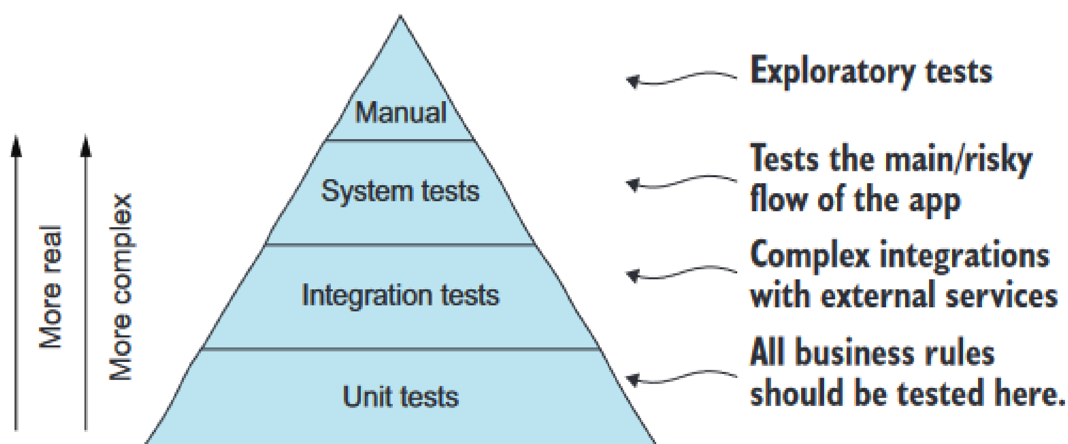
2.5.2 Lasilaatikkotestaus

Lasilaatikkotestauksessa testaaja on vuorostaan tietoinen testattavan ohjelmiston rakenteesta ja toiminnasta. Lasilaatikkotestaus mahdollistaa yksityiskohtaisten testitapausten luomisen, jotka tarkistavat esimerkiksi ehtolauseiden ja silmukoiden oikean toiminnan. Tämä menetelmä auttaa löytämään virheitä, kuten logiikkavirheitä, jotka voivat jäädä huomaamatta mustalaatikkotestauksessa. (Bierig;Brown;Galván;& Timoney, 2022, ss. 143-144)

2.6 Funktionaalinen testaus

Toiminnallinen (funktionaalinen) testaus keskittyy ohjelmiston toiminnallisuuden varmistamiseen. Se on testaustyyppi, jolla varmistetaan, että järjestelmä toimii suunnitellusti ja täyttää sille asetetut toiminnalliset vaatimukset. Toiminnallinen testaus painottaa mitä ohjelmisto tekee ja miten se suorittaa toimintonsa. Esimerkkejä toiminnallisesta testauksesta ovat yksikkötestaus, integraatiotestaus ja systeemitestaus. (Doshi, 2023)

Kattava testausstrategia ei ole pelkästään joukko irrallisia testejä, vaan sen saavuttamiseksi tarvitaan suunnitelmallisuutta ja johdonmukaista lähestymistapaa, joka kattaa erilaiset testausmenetelmät ja -vaiheet koko ohjelmiston elinkaaren ajan. Yksi tunnetuimmista lähestymistavoista testauksen suunnitteluun on Cohnin (s. 312) testauspyramidi, joka tarjoaa pohjan testausstrategian kehittämiseen. Testauspyramidi on keino havainnollistaa miten eri testityyppien tulisi jakaantua suhteessa testien kokonaismäärään (Aniche, 2022, s. 23). Alkuperäinen testauspyramidi on jaettu kolmeen eri osaan, yksikkötesteihin, integraatiotesteihin ja UI-testeihin. Pyramidin pohjalta on kehitetty kuitenkin useita variaatioita, jotka painottavat testauksen osa-alueita eri tavoin.



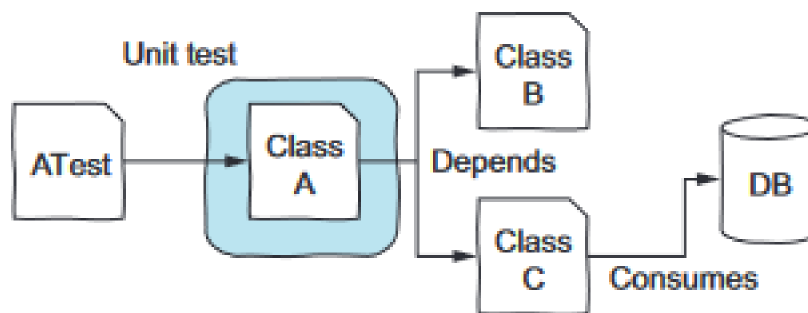
Kuva 5 Anichen versio testauspyramidista (Aniche, 2022, s. 23).

Esimerkiksi Anichen (2022, s. 24) versiossa testipyramidista UI-testit on korvattu systeemitesteillä ja pyramidin huipulla on oma osionsa manuaalitestaukselle. Yksikkötestien määrän tulisi olla suhteessa korkeampi kuin integraatiotestien ja integraatiotestien määrän vastaavasti suurempi kuin systeemitestien. Mitä ylemmäksi pyramidissa mennään, sitä realistisempia, mutta samalla myös monimutkaisempia testitapaukset ovat. Tehokas testausstrategia hyödyntää kaikkia pyramidin tasoja, painottaen kuitenkin matalamman tason testejä. (Kuva 5)

Tällaista strategiaa hyödyntämällä virheiden paikantaminen ja eristäminen helpottuu huomattavasti. Virheen ilmeneminen yksikkötesteissä osoittaa kiistattomasti, että ongelman on oltava testattavassa yksikössä itsessään. Vastaavasti, jos virhe löydetään vasta integraatiotesteissä, se viittaa ongelman liittyvän moduulien yhteistyöhön. (Patton, 2001, s. 112)

2.6.1 Yksikkötestaus

Yksikkötestaus on ohjelmistotestauksen muoto, jossa yksittäisiä yksiköitä tai komponentteja testataan erillään muusta järjestelmästä. Näin varmistetaan, että ne toimivat suunnitellusti. Yksikkö tarkoittaa pienintä mahdollista testattavaa ohjelmiston osaa, joka suorittaa tietyn toiminnon. Yksikkö voi olla metodi, funktio, luokka tai joissain tapauksissa jopa moduuli. (Doshi, 2023)



Kuva 6 Yksikkötesti painottuu yhteen yksikköön, kuten yksittäiseen luokkaan (Aniche, 2022, s. 20)

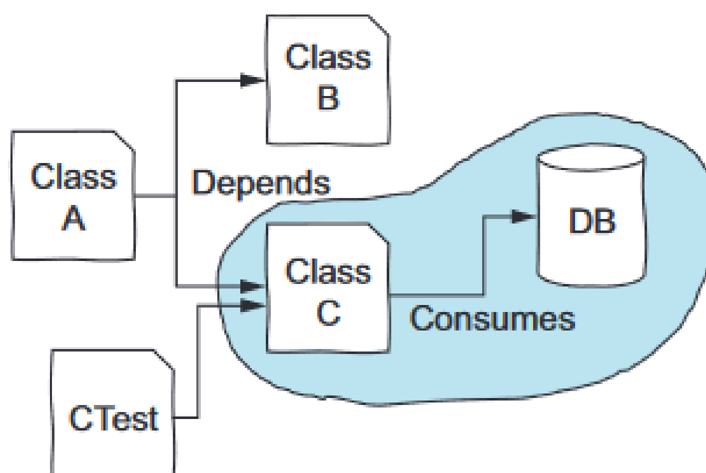
Yksikkötestauksen tavoitteena on testata yhtä yksikköä kerrallaan, mahdollisimman eristettynä muusta ohjelmistosta. Jos luokalla A on riippuvuuksia muihin ohjelmiston osiin, ne täytyy korvata esimerkiksi jäljitelmäolioilla tai laajentamalla testattavaa yksikköä. (Kuva 6)

Yksikkötestien suurin etu on niiden nopeus ja yksinkertaisuus. Yksikkötestin suoritus kestää yleensä vain muutaman millisekunnin, joten niiden avulla on mahdollista testata suuria osa-alueita ohjelmistosta lyhyessä ajassa. Niiden avulla kehittäjä saa jatkuvasti ja nopeasti palautetta ohjelmiston muutoksista. (Aniche, 2022, s. 19) Näin voidaan varmistaa, etteivät muutokset riko jo olemassa olevia ominaisuuksia ja mahdolliset virheet voidaan tunnistaa ja korjata nopeasti. Yksikkötestien ylläpito ja muokkaaminen tarpeen tullessa on myös yksinkertaista. Yksikkötestit testaavat ohjelmistoa antamalla syötteitä testattavalle metodille ja vertaamalla metodin paluuarvoa odotettuun tulokseen. Syötteitä ja odotettuja tuloksia on helppo muokata, jos ohjelmiston vaatimukset ja sitä myötä testattavan metodin toiminta muuttuvat. Lisäksi yksikkötestien kirjoittaminen on helppoa, sillä ne keskittyvät vain yhteen pieneen ohjelmiston osa-alueeseen kerrallaan. Testien tekeminen muuttuu huomattavasti monimutkaisemmaksi, jos testattavalla yksiköllä on riippuvuuksia muihin ohjelmiston osiin, kuten tietokantaan tai käyttöliittymään.

Vaikka yksikkötestauksessa on paljon hyviä puolia, on tärkeää tunnistaa myös sen heikkoudet. Ohjelmistot ovat tyypillisesti monimutkaisia kokonaisuuksia, josta koostuvat useista eri yksiköistä. Pelkkien erillisten yksiköiden testaaminen ei välttämättä paljasta kaikkia mahdollisia virhetilanteita, vaan ne havaitaan vasta myöhemmin esimerkiksi integraatiotestauksen aikana. Yksikkötestit eivät siis anna täydellistä kuvaa ohjelmiston todellisesta toiminnasta.

2.6.2 Integraatiotestaus

Integraatiotestaus on ohjelmistotestauksen vaihe, jossa varmistetaan, että erilliset ohjelmistokomponentit toimivat yhdessä saumattomasti ja muodostavat toimivan kokonaisuuden. Integraatiotestaus suoritetaan yksikkötestausten jälkeen, kun yksittäiset moduulit tai yksiköt on jo testattu erikseen. (VALA Group, 2023)



Kuva 7 Integraatiotestien avulla varmistetaan, että eri komponentit ovat yhteensopivia ja toimivat yhdessä virheettömästi. (Aniche, 2022, s. 21)

Kun yksikkötestit on ajettu, ja niistä löytyneet alemman tason virheet on korjattu, yksittäiset yksiköt yhdistetään muodostaen suurempia moduuleita. Näille uusille moduuleille suoritetaan integraatiotestaus, jonka avulla varmistetaan niiden yhteensopivuus. (Patton, 2001, s. 112) Integraatiotestaus etenee vaiheittain, ja jokaisessa vaiheessa yhdistetään yhä useampia ohjelmiston osia. Tämän prosessin myötä testattavan kokonaisuuden laajuus kasvaa, kunnes merkittävä osa ohjelmistosta on saatu testattua hyväksyttävällä laajuudella. (Kuva 7)

Integraatiotestit keskittyvät kahteen ohjelmiston komponenttiin, joiden integraatiota halutaan testata. Esimerkiksi testi, jonka avulla varmistetaan oman komponenttimme ja ulkoisen tietokannan yhteensopivuus, voisi sisältää seuraavanlaisia vaiheita:

1. Tietokannan käynnistäminen
   Ensimmäinen vaihe on käynnistää tietokanta, johon testattava komponentti ottaa yhteyden. Tämä voi tapahtua manuaalisesti tai automatisoidusti työkalujen avulla ennen varsinaisen testien suorittamista.
2. Yhteys tietokantaan ja tiedon kirjoittaminen
   Kun tietokanta on käynnistetty, testattava ohjelma ottaa siihen yhteyden ja kirjoittaa tietokantaan tietoja.

3. Tietojen lukeminen ja varmistaminen
   Tietojen kirjoittamisen jälkeen samat tiedot luetaan takaisin tietokannasta. Näin varmistetaan, että tiedot on tallennettu oikein ja ne myös pystytään lukemaan ohjelmiston avulla. Lukeminen voi tapahtua esimerkiksi SQL-kyselyiden avulla, tai käyttämällä olio-relaatiomallinnusta (ORM – object-relational mapping), joka abstrahoi tietokannan rakenteen olioiksi, joten ohjelmoijan ei tarvitse huolehtia tietokannan yksityiskohdista tai SQL-kyselyiden luomisesta ja ylläpidosta.
4. Tulosten analysointi ja virheenkorjaus
   Jos kaikki tiedot on saatu tallennettua ja luettua oikein, testi on onnistunut. Muussa tapauksessa testi on epäonnistunut ja virheet on korjattava.

2.7 Järjestelmätestaus

Järjestelmätestauksessa testataan ohjelmistoa kokonaisuutena. Nämä testit suoritetaan yleensä sen jälkeen, kun yksikkö- ja integraatiotestit on saatu suoritettua. Järjestelmätestien tavoitteena voi olla esimerkiksi seuraavien ominaisuuksien testaus (Chauran, 2010, ss. 234-242):

1. Toiminnallisuus: varmistetaan, että kaikki järjestelmän ominaisuudet toimivat halutulla tavalla
2. Suorituskyky: testataan, että järjestelmä on riittävän nopea ja se skaalautuu tarvittaessa
3. Turvallisuus: todennetaan ohjelmiston turvallisuus esimerkiksi luvattomalta pääsyltä ja tietomurroilta

Järjestelmätestaus on tärkeää ohjelmiston laadun varmistamiseksi. Se tarjoaa kattavan näkymän koko järjestelmän toimivuuteen ja auttaa havaitsemaan mahdolliset ongelmat ennen ohjelmiston käyttöönottoa. Tämä testausvaihe varmistaa, että ohjelmisto täyttää sille asetetut vaatimukset ja toimii luotettavasti kaikissa käyttötilanteissa. Ilman järjestelmätestausta riski virheiden ja tietoturva-aukkojen esiintymiselle tuotantoympäristössä kasvaa merkittävästi, mikä voi heikentää käyttäjäkokemusta ja vahingoittaa yrityksen mainetta.

2.8 Manuaalitestaus

Manuaalinen testaus on ohjelmistotestauksen tyyppi, jossa testitapaukset suoritetaan käsin testaajan toimesta ilman automaatiotyökaluja (LambdaTest, 2023). Manuaalitestauksen toteutus ja kattavuus voi vaihdella merkittävästi tilanteen mukaan. Toisinaan se voidaan suorittaa erittäin strukturoidusti, noudattaen tarkasti etukäteen laadittua testausohjelmaa ja -suunnitelmaa. Toisinaan kuitenkin testit voivat olla hyvinkin ad hoc -luonteisia, jolloin testaajat tutkivat järjestelmää joustavasti ilman tiukkoja ennalta määriteltyjä toimintamalleja. Ad hoc -testausta on myös se, kun koodin kirjoittaja suorittaa testejä kehityksen aikana ilman ennalta määriteltyjä suunnitelmia tai testitapauksia. Tällainen testaus tapahtuu spontaanisti ja saattaa sisältää testejä, jotka eivät ole osa virallista testaussuunnitelmaa. Koodin kirjoittaja voi tällöin tutkia koodin toimintaa ja tarkistaa sen toimivuutta ilman tiukkoja suunnitelmia tai ohjeistuksia jo kehityksen alkuvaiheessa.

2.9 Automaattitestaus

Manuaalinen testaus on hidasta, altista virheille ja vaikeasti toistettavaa verrattuna automaattiseen testaukseen. Ihmisen suorittama testaus sisältää väistämättä inhimillisiä virheitä, joita automaattiset, ohjelmistotyökaluilla suoritetut testit eivät tee. Automaattiset testit suoritetaan samalla tavalla joka kerta, mikä eliminoi inhimillisten virheiden mahdollisuuden ja varmistaa tulosten johdonmukaisuuden. (Bierig;Brown;Galván;& Timoney, 2022, s. 230)

Lisäksi automaattiset testit voivat suorittaa suuriakin testisarjoja lyhyessä ajassa, mikä tekee niistä tehokkaan vaihtoehdon toistuville testeille. Tämä lähestymistapa säästää huomattavasti aikaa ja resursseja verrattuna manuaaliseen testaukseen, joka vaatii ihmisten jatkuvaa valvontaa ja panosta. (Bierig;Brown;Galván;& Timoney, 2022, ss. 230-232)

Automaattitestauksesta ja testiautomaatiosta kerrotaan tarkemmin seuraavassa kappaleessa.

3 TESTIAUTOMAATIO

”Testiautomaatio tarkoittaa toimintaa, jossa sopivia työkaluja käyttämällä saadaan tietokone toteuttamaan testitapauksia, ilman ihmisen puuttumista itse testausvaiheeseen.” (VALA Group, 2021)

Testiautomaatio on osa merkittävä osa ohjelmistokehitysprosessia ja ohjelmiston laadunvarmistusta. Sen käytöllä halutaan vähentää manuaalisen testaamisen määrää ja siihen tarvittavia resursseja.

Testitapaukset tulee olla toistettavia, luotettavia, suorituskykyisiä ja modulaarisia. Testiautomaation parhaan mahdollisen hyödyntämisen kannalta on tärkeää, että testitapauksista saatuihin tuloksiin voidaan luottaa. Tarvittaessa testitapaukset pitää olla toistettavissa, jotta vianetsintä on mahdollista. Vianetsintää helpottaa testitapausten modulaarisuus eli se, että testit on jaettu pienempiin kokonaisuuksiin. Tämä helpottaa myös testien ylläpitoa.

Testiautomaation menestyksekkään toteuttamisen kannalta on merkittävää, että valittu strategia ja lähestymistapa ovat sopivia ja niitä noudatetaan. (Jose, 2021, s. 15)

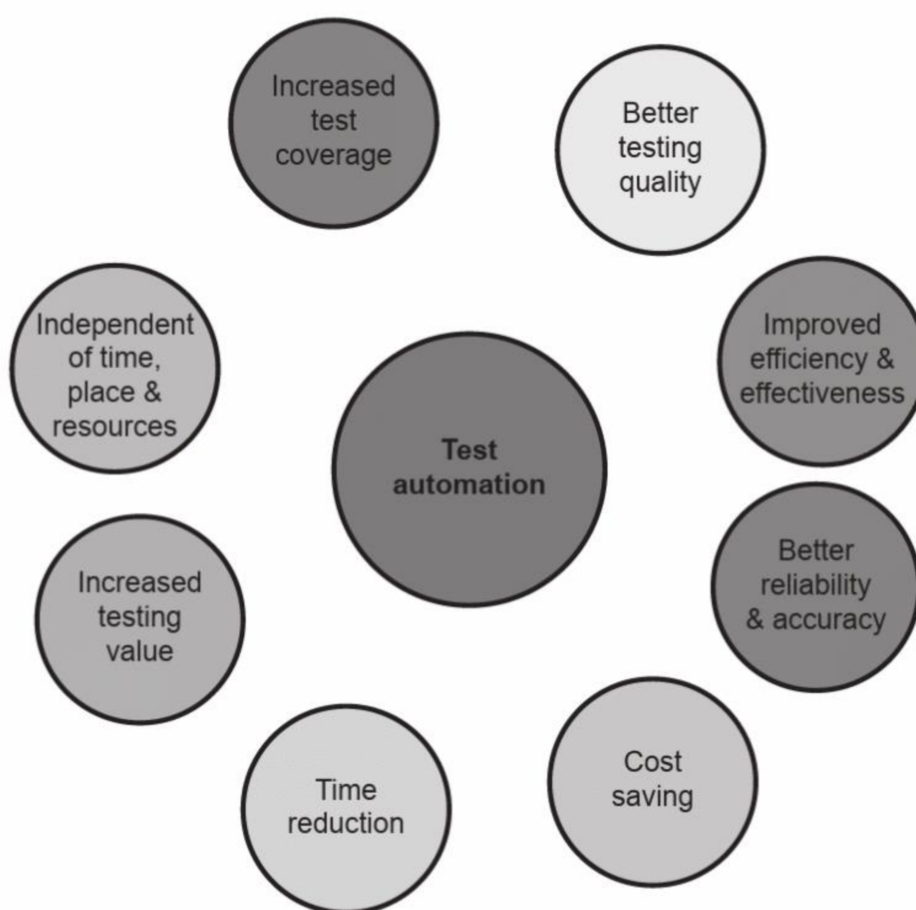
Testauspolitiikalla tarkoitetaan ylemmän tason ajatusta testaamisesta. Siihen kuuluu resurssit ja määritelmä siitä, miten ohjelmaa halutaan testata. Testiautomaatiostrategialla pyritään määrittelemään toimenpiteet, joilla testaus järjestetään. Strategian tulee seurata testauspolitiikassa määriteltyjä suuntaviivoja.

Testaussuunnitelmalla pyritään määrittelemään mahdollisimman tarkasti tavat, miten ohjelmistoa halutaan testata. ”Testaussuunnitelma on projekti- tai tuotekohtainen suunnitelma testausstrategian toteuttamisesta.” (Vala Group, 2023) Suunnitelulla on merkittävä rooli testauksen onnistumisen kannalta.

3.1 Hyödyt

Testiautomaation tuottamat hyödyt näkyvät alla olevassa kuvassa sekä listauksessa (Kuva 8). Josen (Jose, 2021, ss. 10-11) mukaan testiautomaatio tarjoaa seuraavat edut organisaatiolle ja sen asiakkaille ja sidosryhmille:

- Testauksen parempi kattavuus ja laatu
- Parantaa testauksen tehokkuutta
- Parantaa testauksen luotettavuutta ja tarkkuutta. Testiautomaatio on luotettavampaa kuin manuaalinen testaus, koska se vähentää testin suorittamisen aiheuttamia inhimillisiä virheitä.
- Kustannus – ja ajansäästö. Pitkällä aikavälillä hyvin toteutettu testiautomaatio voi vähentää kustannuksia merkittävästi.
- Tehostaa testausta, koska se voi toimia ilman valvontaa.



Kuva 8. Testiautomaation hyödyt (Jose, 2021, s. 11)

3.2 Haasteet

Automaattisen ohjelmistotestauksen järjestämisessä tai toteuttamisessa voi olla haasteellisia tilanteita tai asioita, jotka voivat heikentää ohjelmiston testausprosessia muuten. Haasteet, jotka liittyvät testiautomaation järjestämiseen ovat kuvattu alla olevassa kuvassa (Kuva 9). Jose (Jose, 2021, s. 9) on kirjassaan listannut asioita, jotka aiheuttavat haasteita automaattisessa ohjelmistotestauksessa tai testiautomaation järjestämisessä:

- Vähentää negatiivisen ja tutkivan testauksen kattavuutta. Tutkiva testaus on lähestymistapa, joka parantaa testauksen monipuolisuutta.
- Testiautomaatiotyökalut lisäävät kustannuksia. Kustannuksia syntyy työkalujen ylläpidosta ja kehityksestä.
- Testiautomaation tuottamien tulosten analysointi vaatii resursseja.
- Automatisoitu ohjelmistotestaus ei sovi osaksi lyhyiden projektien testausprosessia.
- Testattavalta sovellukselta vaaditaan vakautta, jotta sille voidaan järjestää automatisoitua ohjelmistotestausta.
- Testaustyökalujen viat voivat aiheuttaa ongelmia testiautomaatiossa.

Kuva 9. Testiautomaation haasteet (Viljamaa, 2024)

3.3 Työkalut ja tekniikat

Testiautomaatiossa käytössä olevien työkalujen valintaan vaikuttavat useat tekijät. Valittujen tekniikoiden tulee tukea testaustapaa. Testiautomaatiotyökalut ovat ohjelmistokokonaisuuksia, joilla toteutetaan sovelluksen automaattinen testausprosessi. Käytännössä työkaluja ja testiautomaation toteutustapoja on monia. Avoimen lähdekoodin ratkaisut voivat olla kustannusten kannalta parempia, vaikka kaupalliset testiautomaatiotyökalut voivat tuoda lisäarvoa testaukseen ja sen luotettavuuteen.

”Kun on päätetty mitä automatisoidaan, on työkalujen valinta kriittistä.” (VALA Group, 2021) Oikeanlaisten ja toimivien työkalujen valinta on merkittävässä roolissa testauksen onnistumisen kannalta. Vääränlaiset tai sopimattomat työkalut voivat pahimmassa tapauksessa johtaa testausprosessin laatuun huonontavasti. Testiautomaatio vähentää manuaalisen testauksen tarvetta, mutta työkalut vaativat myös henkilöresursseja ylläpitoon. Työkalujen testaus ja määrittely on osa testauksen suunnittelua ja strategian luontia. (VALA Group, 2021)

3.3.1 Testaustyökalut

Testauksessa käytettävät ohjelmistokehykset ohjelmistokokonaisuuksia, joilla voidaan hoitaa joitakin osia testiautomaatiosta. Oikeanlaisen ohjelmistokehyksen valinnalla on merkittävä rooli testauksen

onnistumisen kannalta. ”Ne voivat vähentää ylläpitokustannuksia ja testaukseen käytettyä vaivannäköä sekä tarjota paremman sijoitetun pääoman tuoton (ROI) laadunvarmistustiimeille, jotka haluavat optimoida ketterät prosessinsa.” (SmartBear Software, ei pvm)

Testiautomaatio ohjelmistokehyksen käyttö tarjoaa erilaisia hyötyjä, jotka parantavat testausprosessia. Ohjelmistokehyksen valinta tulee tehdä suhteessa testattavaan sovellukseen.

Yleisimpiä automaattisessa ohjelmistotestauksessa käytettäviä ohjelmistokehyksiä ovat mm. Robot Framework, Selenium, Appium ja Espresso.

Robot Framework on avoimen lähdekoodin automaatio ohjelmistokehys. Yleisesti sitä käytetään robottisessa testausprosessissa ja testiautomaatiossa. Robot Frameworkia ylläpitää Robot Framework Foundation. (Robot Framework, 2024) Robot Framework on avainsanapohjainen ohjelmistokehys. Se tarkoittaa sitä, että testit koostuvat avainsanoista, jotka ovat funktioita, joita suoritetaan järjestyksessä. Robot Frameworkin etuna on se, että se on integroitavissa monenlaisiin projekteihin. (VALA Group, 2021) Sen etuna on myös syntaksin helppo luettavuus, joka helpottaa testien ymmärrettävyyttä ilman suurta määrää asiantuntemusta.

Selenium on työkalu, jolla emuloidaan käyttäjän toimintaa selaimella. Selenium on avoimen lähdekoodin työkalu, jolla hoidetaan web pohjaisten sovellusten testiautomaatiota. (VALA Group, 2021) Selenium on laajalti käytetty ja suosittu työkalu, joka tekee siitä hyvin tuetun.

Appium on testiautomaatiotyökalu, joka tukee eri alustojen käyttöliittymän automatisointia. Se on yleisesti käytössä mobiilisovellusten testiautomaatiossa. Lisäksi se pyrkii tukemaan eri ohjelmointikielillä kirjoitettua automaatiokoodia. Tuetut kielet ovat JavaScript, Java, Python. (Appium, 2024) Vala Groupin testiautomaatio-oppaan mukaan Appium sopii hyvin blackbox-testaukseen. (VALA Group, 2021)

Espresso on ohjelmistokehys, jolla voidaan tehdä käyttöliittymätestausta Androidille. Se julkaistiin vuonna 2013 Googlen toimesta. ”Sen tärkeimpiä ominaisuuksia on se, että testitoiminnot synkronoidaan automaattisesti käyttöliittymän kanssa.” (Nabil, ei pvm)

3.3.2 Ohjelmointikielet

Testiautomaation järjestäminen ja ylläpito vaatii asiantuntemusta ja osaamista ohjelmoinnista, ohjelmointikielistä ja erilaisista ohjelmistokehyksistä.

Yleisesti testiautomaatiossa käytettäviä ohjelmointikieliä ovat: Python, Perl, Java, C#, Ruby, PHP ja JavaScript. (Jose, 2021, s. 148) Ohjelmointikielen valintaan vaikuttavat testattavan ohjelmiston lisäksi testausympäristö ja sen vaatimukset.

Java on laajalti käytetty ohjelmointikieli testiautomaatiossa. Sen etuina ovat vakaus ja luotettavuus, jonka takia se on suosittu valinta erilaisissa testausympäristöissä ja tarpeissa. (AnAr Corporate, 2023)

Tieturi (Tieturi Oy, 2023) on kuivaillut Javaa seuraavasti: ”Java on monipuolinen ohjelmointikieli, joka mahdollistaa tehokkaiden ja luotettavien sovellusten kehittämisen.” Javan merkittävä etu on se,

että se on alustariippumaton. Tämä tarkoittaa sitä, että sitä voidaan käyttää missä tahansa käyttöjärjestelmässä.

C# (CSharp) - Ohjelmointikieli voi luoda suurta lisäarvoa testiautomaatiossa. Sen etuina voidaan pitää sitä, että se mahdollistaa erilaisten sovellusten testaamisen eri alustoilla. Tällaisia alustoja ovat esimerkiksi Windows, iOS ja Android. Se on Microsoftin tukema, joka tekee siitä luotettavan ja hyvän ratkaisun pitkällä aikavälillä. C# on käännettävä ja vahvasti tyypitetty ohjelmointikieli. (AnAr Corporate, 2023)

Python on laajalti testiautomaatiossa käytettävä avoimen lähdekoodin ohjelmointikieli. ”Se tarjoaa erilaisia ohjelmistokehyksiä yksikkö-, päästä-päähän- ja integraatiotestaukseen, joita käytetään usein testiautomaatiossa.” (AnAr Corporate, 2023) Python on tulkattava ohjelmointikieli, jonka syntaksia pidetään helposti luettavana. Python on myös testiautomaation ulkopuolella yksi suosituimmista ohjelmointikielistä.

4 CICD JA DEVOPS

4.1 Johdanto

DevOps tarkoittaa ohjelmistokehityksessä käytössä olevaa toimintatapaa, jolla pyritään lisäämään organisaation kykyä toimittaa ja kehittää sovelluksia suuremmalla nopeudella. DevOps:n tarkoituksena on vähentää sovellusta kehittävän organisaation jakoa, sekä parantaa organisaation osien kommunikointia toistensa kanssa. (Amazon, ei pvm) Sana DevOps on lyhenne sanoista ”development” ja ”operations”. Siihen kuuluu toimintamallien lisäksi lukuista joukko työkaluja, kuten jatkuvan integroinnin ja jatkuvan toimituksen työkalut.

Synopsysin (Synopsys, ei pvm) mukaan DevOps:n luomat edut ovat mm. nopeus ja nopea toimitus, parantaa luotettavuutta, yhteistyötä ja turvallisuutta. DevOps:lla voidaan saavuttaa merkittäviä hyötyjä ohjelmistokehitysprosessin aikana. Organisaation ja sen osien kommunikaatio ja viestintä toistensa kanssa on avain asemassa projektin onnistumisen kannalta. Alun perin DevOps:lla on pyritty välttämään niin sanottua siiloutumista eli tiimien erkanemista toisistaan.

DevOps on yleinen ja suosittu palveluiden kehitysmalli. Sillä voidaan saavuttaa merkittäviä hyötyjä, mutta on olemassa tilanteita, joihin se ei sovellu. DevOps – mallin toteuttamiselle haasteita voivat tuottaa resurssit. ”DevOpsin haasteita ovat testauksen automatisoinnin vaatimat investoinnit ja menetelmän käyttöönoton vaatimat resurssit”. (IteWiki, ei pvm) DevOps:n vaatimukset resurssoinnille voivat tuottaa lyhyellä aikavälillä kustannuksia, mutta pitkällä aikavälillä ne voivat tuoda lisäarvoa ja säästöjä yritykselle.

Vaikka DevOps voi tuoda merkittäviä hyötyjä organisaatiolle, on tilanteita, joihin se ei sovellu. Tällaisissa tilanteissa DevOps ei tuo merkittävää hyötyä tai sen käyttöönotto ei ole tarkoituksenmukaista. Yrityksen tulee tiedostaa oman toimialueensa vaatimukset ennen DevOps:n käyttöönoton suunnittelua. Joissain tilanteissa esimerkiksi automatisointi voi olla tarkoituksenmukaista, vaikka yritys ei hyötyisikään DevOps – toimintamallista.

Tilanteita, joissa perinteinen ohjelmistokehitysmalli voi toimia paremmin kuin DevOps ovat artikkelin (Chickowski, ei pvm) mukaan:

- Yritys ei tarvitse ohjelmiston jatkuvaa päivittämistä ja kehittämistä
- Yritys toimii korkeasti säännellyllä alalla
- Yritys ei saa merkittävää liiketoiminnallista hyötyä DevOps – toimintamallista
- Yrityksessä tapahtuu fuusioita tai yritysostoja

Organisaation kokemus ja koko vaikuttaa merkittävästi siihen, miten DevOps – toimintamallin käyttöönotto onnistuu.

”DevOps:n tulevaisuus on jännittävä, sillä se lupaa mullistaa organisaatioiden ohjelmistokehitysprosessin ympärin maailman”. (JumpGrowth, 2023) Sen kannalta yksi tärkeimmistä trendeistä tulevaisuudessa tulee olemaan tekoäly ja koneoppiminen. Näillä teknisillä ratkaisuilla voidaan automatisoida ohjelmistokehityksen yleisimpiä ja toistuvia toimintoja. Tällä voidaan vapauttaa resursseja ja saada suuria säästöjä.

DevOps – toimintamallia voidaan pitää nykyään merkittävänä ja tärkeänä osana ohjelmistokehitystä. Se lyhentää kehityssykliä ja lisää yrityksen kilpailukykyä. Toimintamallin omaksuminen lisää tehokkuutta, joka mahdollistaa uusien tekniikoiden käyttöönoton mahdollisimman lyhyessä ajassa. DevOps – toimintamallin mukainen tehostaminen perustuu erityisesti organisaation sisäisen kommunikaation ja yhteistyön paranemiseen.

4.2 DevOps periaatteet

”DevOps on digitaalisten palveluiden kehitys- ja tuontantomalli, jonka periaatteita ovat ketterä kehitys, jatkuva integraatio (continuous intergration) ja jatkuva toimitus (continuous delivery).” (IteWiki, ei pvm) Ketterällä kehityksellä tarkoitetaan tuotantomallia, joka perustuu aktiiviseen viestintään, käyttäjien palautteeseen ja siihen reagointiin sekä lyhyisiin iteraatioihin eli jaksoihin. Sen etuja ovat erityisesti reaktiivisuus ja tehokkuus. Ketterän kehityksen projektinhallinnassa käytettäviä viitekehyksiä ovat esimerkiksi Scrum ja Lean. Jatkuvaa kehitystä ja jatkuvaa integraatiota käsitellään jäljempänä tässä kappaleessa.

Näiden periaatteiden lisäksi keskeistä DevOps – toimintamallin toteuttamisessa on automatisointi. Tällä tarkoitetaan testauksen automatisointia ja ympäristöjen automatisoitua konfigurointia. Automatisoinnilla pyritään säästämään resursseja, mutta samalla parantamaan tehokkuutta ja tarkkuutta läpi ohjelmistokehitysprosessin. Automatisoitua ohjelmistotestausta on käsitelty kappaleessa 3.

Jokaisella ohjelmistokehitysprojektilla on omat tarpeensa ja vaatimuksensa, joka vaikuttaa siihen valitaanko tuotantomalliksi DevOps vai perinteisemmät ohjelmistokehityksen mallit. Vesiputousmalli on yleinen perinteisen ohjelmistokehityksen malli. Keskeisiä eroja perinteisen ohjelmistokehityksen ja DevOpsin välillä ovat artikkelin (Dunster, ei pvm) mukaan:

- Julkaisurytmi on huomattavasti hitaampi perinteisessä ohjelmistokehityksessä
- Saadun datan hyödyntäminen on vähäisempää perinteisessä ohjelmistokehitysmallissa
- Riskienotto on vähäisempää perinteisessä ohjelmistokehityksessä
- Kustannusten leikkaus on vaikeampaa perinteisessä ohjelmistokehityksessä
- Perinteisessä ohjelmistokehityksessä henkilö toteuttaa tehtävän alusta loppuun, joka voi aiheuttaa laatuongelmia

Perinteisille ohjelmistokehitysmalleille tyypillinen hitaampi julkaisurytmi voi olla riskialttiimpaa kuin DevOps – toimintamallin mukaiset lyhyet julkaisusyklit. Mahdolliset ongelmat ja niiden korjaaminen vaikeutuu suurien julkaisujen yhteydessä, joka heikentää tehokkuutta.

DevOps – toimintamallin mukaisessa ohjelmistokehityksessä datan hyödyntäminen ja sen mukainen reagointi on helpompaa automatisoitujen prosessien ansioista.

Riskienoton kannalta eroavaisuus tulee siinä, että DevOps - toimintamallissa suositaan riskienottoa ja siinä omaksutaan niin kutsuttu aikaisen epäonnistumisen kulttuuri. Siinä pyritään tunnistamaan tilanteet, joissa epäonnistuminen on väistämätön.

Perinteisissä ohjelmistokehitysmalleissa onnistumiseen pyritään vähentämällä kustannuksia. Tämän takia perinteisen ohjelmistokehityksen toimintamallissa tehostaminen voi olla merkittävästi vaikeampaa kuin DevOps – toimintamallissa.

4.2.1 Työkalut ja teknologiat

DevOps–käytäntöihin liittyy toimintakulttuurin lisäksi paljon erilaisia työkaluja. GitLabin (GitLab, ei pvm) mukaan työkalut voidaan kategorisoida seuraavasti:

- Kommunikaatio ja yhteistyö
- Jatkuva integraatio ja käyttöönoton automaatio
- Testiautomaatio
- Versiohallinta
- Konttien hallinta
- Valvonta
- Konfiguraation hallinta
- Artefaktien hallinta

Kommunikaatiotyökalut ovat nimensä mukaisesti työkaluja, joilla organisaatiossa pidetään yhteyttä ja vaihdetaan tietoa. Viestintä on avainasemassa DevOps:n mukaisen ohjelmistokehityksen toteuttamisen kannalta. Tällaisia työkaluja ovat esimerkiksi Slack ja Microsoft Teams.

Versiohallintatyökalut ovat työkaluja, jotka mahdollistavat samanaikaisen kehittämisen samojen tiedostojen kanssa. Versiohallinnassa käytetyin tekniikka on Git. Versiohallinta on välttämätöntä, kun halutaan työskennellä useamman henkilön ryhmissä muokaten samaa ohjelmistokoodia.

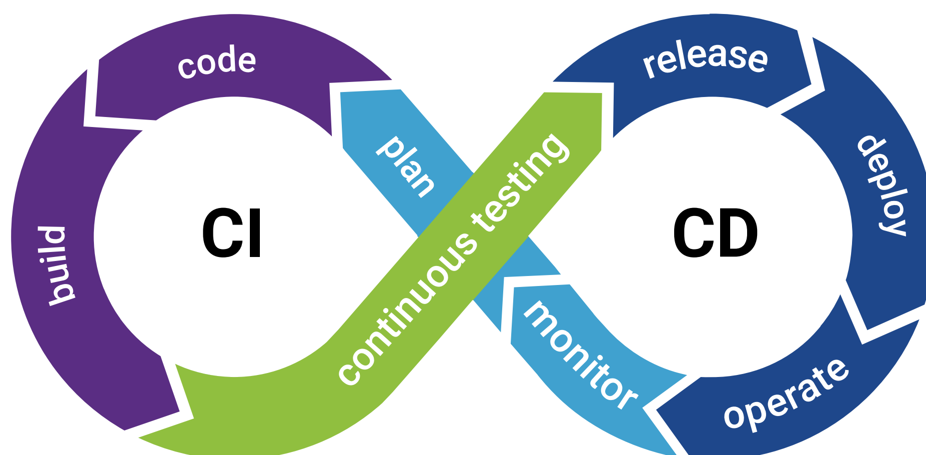
Muita DevOps – toimintamalliin liittyviä työkaluja ovat erilaiset automatisointityökalut, joilla voidaan automatisoida ohjelman rakennus, testaus ja julkaisu. Tällaisia automatisointiin liittyviä työkaluja ovat Jenkins ja GitLab CI.

4.3 Jatkuva integraatio ja jatkuva toimitus

Jatkuvalla integraatiolla tarkoitetaan toimintatapaa ohjelmistokehityksessä, jossa jokainen ohjelmistoon tehtävä muutos testataan automaattisesti osana jo olemassa olevaa järjestelmää. Jatkuvalla toimituksella tarkoitetaan menetelmää, joka mahdollistaa ohjelmiston nopean julkaisun. Jatkuva integraatio ja jatkuva toimitus kuvataan usein kuvan mukaisesti syklisenä (Kuva 10).

Jatkuva integraatio varmistaa, että muutokset integroituvat ja toimivat osana järjestelmää. ”Jatkuvassa integraatiossa koodin vahvistamisen jälkeen ohjelmisto rakennetaan ja testataan välittömästi” (Hamilton, 2024)

Jatkuva toimitus mahdollistaa ohjelmistojen säännöllisen ja luotettavan julkaisun erityisesti automatisoinnin avulla. Toistuvat ja aikaa vievät prosessit pyritään automatisoimaan mahdollisimman pitkälle. Ohjelmistotestaus on esimerkki tällaisesta aikaa vievästä prosessista.



Kuva 10. Jatkuvaa integraatiota (CI) ja jatkuvaa toimitusta (CD) kuvataan usein syklinä.

4.3.1 Hyödyt

Jatkuva integraatio ja jatkuva toimitus voi tuoda merkittäviä kaupallisia hyötyjä yritykselle. Artikkelissa (Kariappa, 2024) listattuja hyötyjä, jotka tuovat hyötyjä yritykselle ovat:

- Nopeampi käyttöönotto
- Parantaa koodin laatua
- Helpottaa ongelmien paikantamisen ohjelmakoodista
- Resurssisäästöt
- Vähentää manuaalista työtä
- Palautteen huomioiminen helpottuu
- Mahdollistaa nopeamman palautumisen ongelmista
- Lisää läpinäkyvyyttä koko kehitysprojektin kannalta
- Helpottaa ongelmien korjausta myös ei-kriittisissä ongelmissa

Jatkuva integraatio ja jatkuva toimitus lyhentää ohjelmistotuotteen markkinoille tuloaikaa erityisesti automatisoinnin ansiosta. Se helpottaa myös koodimuutosten aiheuttamien ongelmien paikannuksen, koska muutoksien koko voi olla pienempi kuin perinteisessä ohjelmistokehitysmallissa. Manuaalisen työn väheneminen tuo pitkällä aikavälillä suuria resurssi- ja kustannussäästöjä.

5 KUORMANJAON TEORIA JA ALGORITMIT

5.1 Johdanto

Kuormanjako on tekniikka, jota käytetään työmäärän jakamiseen useiden laskentaresurssien tai palvelimien kesken. Kuormanjaon tavoitteena on optimoida resurssien käyttöä ja estää yksittäisen resurssin tai palvelimen ylikuormittuminen (Umbraco, julkaisuaika tuntematon). Vaikka kuormanjako liitetään usein palvelimiin, joilla pyritään tasapainottamaan verkkopalveluihin kohdistuvaa liikennettä, sen periaatteita voidaan soveltaa myös laajemmin.

Kuormanjaon toteutukseen on olemassa useita algoritmeja, sillä ei ole olemassa yhtä universaalia tapaa, joka toimisi kaikissa tilanteissa. Algoritmin valintaan vaikuttaa esimerkiksi jaettavien kuormien luonne (ovatko ne esimerkiksi riippuvaisia toisistaan) ja kuorman suorittavan resurssin ominaisuudet.

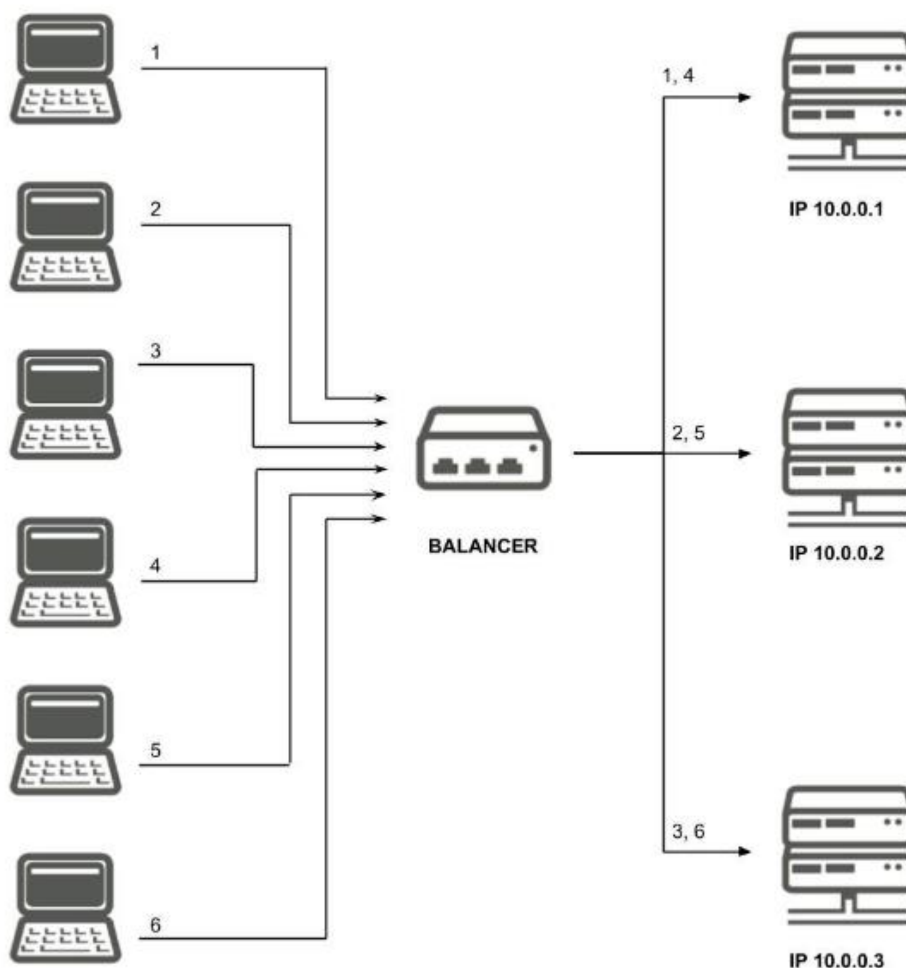
5.2 Staattinen kuormanjako

Staattisessa kuormanjaossa kuormat jaetaan saatavilla oleville resursseille ennen itse suorituksen alkua. Suurimmassa osassa tapauksista voidaan hyödyntää etukäteen tiedettyä informaatiota jaettavista kuormista ja kuorman suorittavista järjestelmistä (Chengzhong & Francis, 1997, s. 6) . Näitä tietoja ovat mm. kuorman tyyppi, odotettu ajoaika sekä resurssin kapasiteetti ja suorituskyky.

Staattisten kuormanjaon algoritmien suurin etu on se, ettei niiden suoritus aiheuta ylimääräistä laskenta-aikaa itse suorituksen aikana. Koska jako voidaan tehdä jo ennen suorituksen aloittamista, se sopii hyvin tilanteisiin, joissa kuormien kestot ovat ennustettavissa. Vastaavasti tilanteissa, joissa kuormien kestot vaihtelevat suuresti tai niitä ei voida ennustaa, staattiset kuormanjaon algoritmit eivät välttämättä ole paras mahdollinen ratkaisu. Tällöin dynaamiset kuormanjaon algoritmit, jotka voivat mukauttaa kuormien jakautumista reaaliajassa, ovat usein parempi vaihtoehto. (Chengzhong & Francis, 1997, ss. 6-7)

5.2.1 Kiertovuorottelu

Kiertovuorottelumenetelmässä (engl. round robin) tehtävät jaetaan resursseille vuorotellen, mikä tarkoittaa, että jokainen resurssi saa vuorollaan suoritettavakseen tehtävän (Padilha, 2018, ss. 5-6).



Kuva 11 Kuormanjakaja tasaa tulevat pyynnöt kolmen palvelimen kesken kiertovuorottelumenetelmällä. (Padilha, 2018, s. 6)

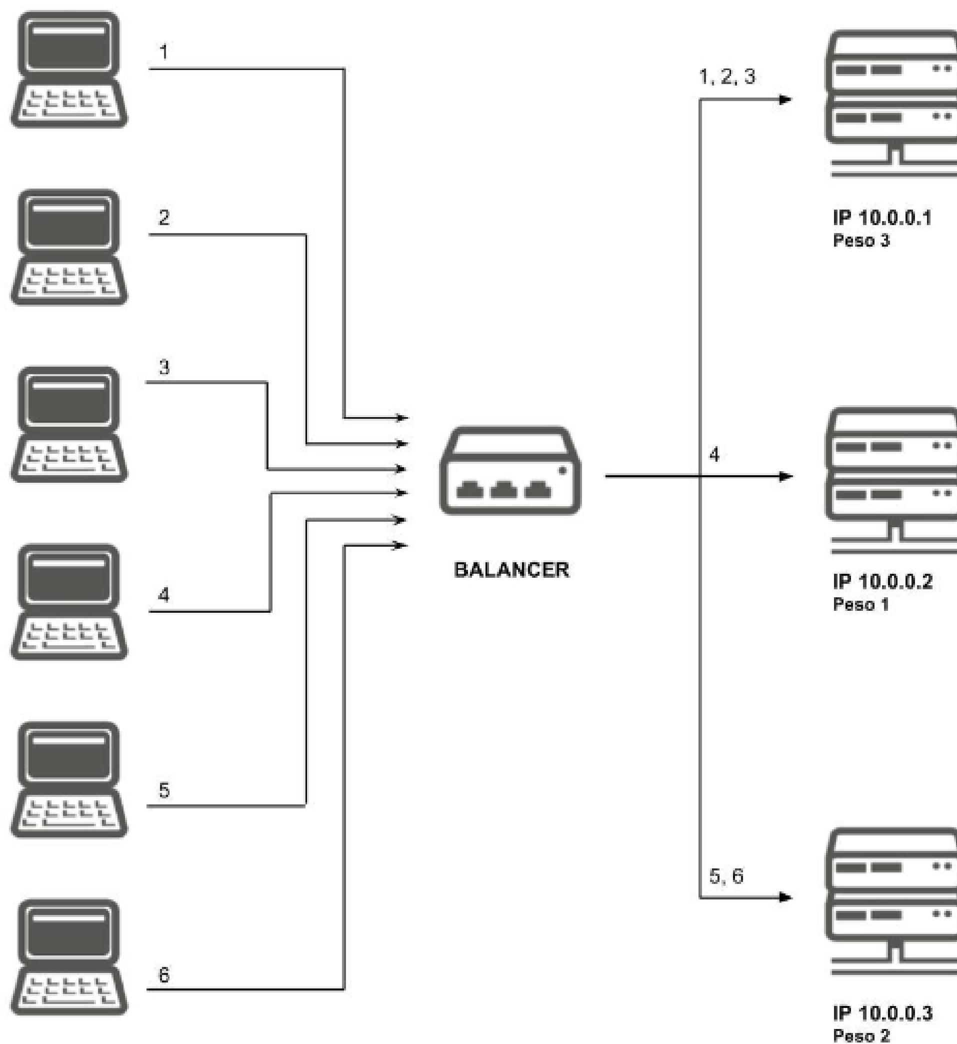
Kuva havainnollistaa tilannetta, jossa kuormanjakajalle tulee kuusi pyyntöä. Kiertovuorottelualgoritmin mukaisesti ensimmäinen pyyntö ohjautuu ensimmäiselle palvelimelle, toinen toiselle ja kolmas pyyntö kolmannelle palvelimelle. Kun kaikille palvelimille on jaettu yksi pyyntö, kierros alkaa alusta ja pyynnöt ohjataan palvelimille samassa järjestyksessä. Neljäs pyyntö ohjautuu siis ensimmäiselle palvelimelle, viides toiselle ja kuudes kolmannelle. (Kuva 11)

Kiertovuorottelumenetelmä ei ota huomioon resurssien välisiä eroja, joten se toimii parhaiten silloin, kun kaikki resurssit ovat samankaltaisia. Jos resurssien suorituskyvyssä tai muissa ominaisuuksissa on merkittäviä eroja, se voi johtaa pullonkauloihin ja heikentää koko järjestelmän tehokkuutta. Tällöin on suositeltavaa käyttää algoritmia, joka ottaa nämä erot huomioon. (Padilha, 2018, s. 6)

5.2.2 Painotettu kiertovuorottelu

Painotettu kiertovuorottelualgoritmi on kehittyneempi versio perinteisestä kiertovuorottelumenetelmästä, joka ottaa huomioon resurssien erilaiset ominaisuudet tehtävien jakamisessa. Jokaiselle resurssille määritetään paino, joka heijastaa sen suorituskykyä tai kapasiteettia. Tämän mekanismin

ansiosta tehtävät jakautuvat resurssien välille tehokkaasti, mikä johtaa järjestelmän suorituskyvyn kasvuun ja pullonkaulojen vähenemiseen. (Padilha, 2018, ss. 6-7)



Kuva 12 Esimerkki painotetusta kiertovuorottelualgoritmista (Padilha, 2018, s. 7).

Kuvan tilanteessa palvelimien painot ovat 3:1:2, joten ensimmäinen palvelin käsittelee kolme ensimmäistä pyyntöä, toinen palvelin yhden pyynnön ja kolmas palvelin kaksi viimeistä pyyntöä. Tämän jälkeen kierros alkaa taas alusta, eli seitsemäs pyyntö ohjautuu jälleen ensimmäiselle palvelimelle. Jos painot on onnistuttu määrittämään oikein, voimme päätellä, että ensimmäisellä palvelimella on paras kyky käsitellä kuormia, kolmannella palvelimella toisiksi paras, ja toisella palvelimella huonoin. Jos määritetyt painot eivät vastaa todellista järjestelmien suorituskykyä, se johtaa pullonkauloihin ja huonoon resurssien hyödyntämiseen. Järjestelmää on seurattava jatkuvasti ja painoja muutettava, jos huomataan, ettei järjestelmä suoriudu tehtävistä halutulla tavalla. (Kuva 12)

5.3 Dynaaminen kuormanjako

Dynaamisessa kuormanjaossa kuormanjakaja ottaa huomioon järjestelmän nykyisen tilan ennen tehtävien jakamista resursseille. Tämä tila voi olla esimerkiksi tämänhetkinen prosessorin kuormitustaso, tai jo käsittelyn alla olevien tehtävien määrä. Se on joustavampi lähestymistapa kuormitusten jakamiseen verrattuna staattiseen kuormanjakoon. Kuormien jakaminen tapahtuu reaaliajassa, jonka

ansiosta kuormanjakaja voi tarkkailla resurssien sen hetkistä tilaa ja saapuvien kuormien ominaisuuksia. Nämä tiedot mahdollistavat päätöksien teon älykkäästi myös suorituksen aikana, parantaen järjestelmän suorituskykyä. (Chengzhong & Francis, 1997, ss. 7-8)

5.3.1 Hajautettu dynaaminen kuormanjako

Hajautetussa mallissa kaikki järjestelmän resurssit osallistuvat dynaamisen kuormanjakoalgoritmin suorittamiseen, ja vastuu kuormituksen tasapainottamisesta on jakautunut niiden kesken. Tässä mallissa resurssit kommunikoivat keskenään ja tekevät päätöksiä siitä, mikä resurssi ottaa vastuulleen tietyn kuorman suorittamisen. Resurssit hoitavat siis samanaikaisesti molempia tehtäviä, kuormanjakoa sekä itse työn suorittamista. Jokainen resurssi valvoo omaa työkuormaansa ja järjestelmän kokonaiskuormaa. Jos resurssi havaitsee olevansa ylikuormitettu, se voi tehdä päätöksen kuormituksen tasapainottamiseksi esimerkiksi jakamalla tehtävän toiselle resurssille. Tällaista dynaamisen kuormanjaon mallia voidaan käyttää esimerkiksi hajautetussa laskennassa. (Alakeel, 2009, s. 155)

Hajautetuissa järjestelmissä kuormanjakoalgoritmit tuottavat yleensä enemmän viestiliikennettä, kuin keskitetyissä järjestelmissä. Tämä johtuu siitä, että jokaisen resurssin on mahdollisesti kommunikoitava kaikkien muiden järjestelmän resurssien kanssa tehdäkseen omat päätöksensä kuormanjakoon liittyen. Merkittävänä etuna on kuitenkin se, että yhden tai useamman resurssin vioittuessa koko järjestelmän toiminta ei pysähdy kokonaan, vaan se johtaa ainoastaan osittaiseen suorituskyvyn heikkenemiseen. (Alakeel, 2009, s. 155)

Hajautetun järjestelmän tehokkuuden optimointiin pyritään minimoimalla resurssien välinen kommunikaatio ja maksimoimalla niiden itsenäinen päätöksenteko. Mitä enemmän resurssien välillä on kommunikaatiota, sitä enemmän järjestelmän suorituskyky heikkenee. (Alakeel, 2009, s. 155)

5.3.2 Ei-keskitetty dynaaminen kuormanjako

Ei-keskitetyssä kuormanjaossa yksi tai useampi resurssi on vastuussa muiden resurssien kuormituksen hallinnasta ja tehtävien jakamisesta. Toisin kuin hajautetussa järjestelmässä, kaikki resurssit eivät kuitenkaan osallistu kuormanjakoon, vaan osa niistä toimii ainoastaan tehtävien suorittajina. (Alakeel, 2009, s. 155)

Ei-keskitettyä dynaamista kuormanjakoa voidaan suorittaa joko keskitetysti tai puolihajautetusti. Keskitetyssä mallissa kuormanjaon suorittaa vain yksi resurssi. Vastaavasti puolihajautetussa kuormanhallintajärjestelmässä resurssit jaetaan ryhmiin, ja jokaisessa ryhmässä yksi resurssi toimii kuormanjakajana koko ryhmälle. Koko järjestelmän kuormanhallinnasta vastaavat jokaisen ryhmän kuormanjakajina toimivat resurssit. Nämä resurssit kommunikoivat keskenään ja koordinoivat järjestelmän kokonaiskuormaa. (Alakeel, 2009, s. 155)

6 TYÖN SUUNNITTELU

6.1 Vaatimusmäärittely

Sovelluksen vaatimusmäärittely on tärkeä osa ohjelmiston suunnittelua. Vaatimusmäärittelyssä käydään läpi ne tavoitteet, jotka ohjelmiston tulee saavuttaa. Kun ohjelmiston vaatimukset ja tavoitteet on listattu, se luo selkeyttä toteutusvaiheeseen. Vaatimusmäärittelyllä voidaan vähentää väärinkäsityksiä ja epätietoisuutta toteutukseen liittyen. (Metatavu Oy, ei pvm)

Toiminnalliset vaatimukset ovat ohjelmiston toimintaan ja sen tarjoamiin palveluihin liittyviä vaatimuksia. Siihen kuuluvat näiden palveluiden lisäksi myös poikkeamien hallinta ja virheenkäsittely. Toiminnallisia vaatimuksia määritellessä edetään suurista kokonaisuuksista kohti pieniä osia. (Taina, 2005)

Ei-toiminnalliset vaatimukset pyrkivät määrittelemään erilaisia rajoituksia, jotka liittyvät toiminnallisiin vaatimuksiin. Ne eivät ota kantaa ohjelmiston tarjoamiin palveluihin, mutta ohjaavat ja asettavat ehtoja toiminnallisille vaatimuksille. Ei-toiminnalliset vaatimukset voivat liittyä ohjelmiston kokoon, käytettävyyteen, nopeuteen ja luotettavuuteen.

Toiminnallisten- ja ei-toiminnallisten vaatimusten lisäksi ohjelmistolle voidaan asettaa implisiittisiä vaatimuksia. ”Implisiittiset vaatimukset ovat ominaisuuksia, joiden oletetaan olevan kaikilla kehitettävän tyyppisillä laadukkailla ohjelmistoilla”. (Taina, 2005) Näitä ei kuitenkaan yleisesti listata osaksi vaatimusmäärittelyä.

Opinnäytetyössä toteutettavan palvelimen vaatimusmäärittelyssä tulee kiinnittää toiminnallisten vaatimusten lisäksi huomiota sen käyttäjiin, tarkoitukseen ja toteutukseen liittyviin vaatimuksiin. Palvelimen tarkoituksena on parantaa automaattista ohjelmistotestausta. Tarkoituksena on saada testausprosessista mahdollisimman kattava. Päätavoitteena palvelimella on parantaa julkaistavan ohjelmiston laatua. Palvelimella voidaan huolehtia, että julkaistaville ohjelmistoversioille ajetaan mahdollisimman paljon testitapauksia. Toteutettavan palvelimen tarkoituksena on jakaa optimaaliset testitapauskombinaatiot erilaisten testausympäristöjen välillä. Lisäksi palvelimen tarkoituksena on päästä eroon manuaalisesta työstä, jotka liittyvät testitapausten jaotteluun testiympäristöjen välillä. Tällä hetkellä Ponsse Oyj:n testiautomaatiossa testitapaukset on määritelty ennalta sovitulla tavalla hyödyntäen Robot Frameworkin argumenttitiedostoja.

Toteutettavan palvelimen tärkein toiminnallinen vaatimus on, että sen tehtävänä on hoitaa testien jaottelu ja testiajon keston laskenta. Jokainen testiympäristö hakee sille asetetut ja lasketut automaatiotestit ennen testauksen aloittamista. Palvelinta käytetään CI/CD – Agentin ja Python-skriptin avulla. CI/CD – Agentti on ohjelmisto, jolla suoritetaan erilaisia tehtäviä liittyen ohjelmiston kääntämiseen tai testaamiseen. Tämän lisäksi palvelimen yksi tärkeimmistä toiminnallisista vaatimuksista on se, että se ylläpitää tietokantaa testituloksista ja testiajojen kestosta. Se on tärkeää, koska tietokanta on pohjana koko palvelimen toiminnalle.

Palvelimen toiminnallisiin vaatimuksiin lukeutuu sen kyky käsitellä http – pyynnössä annetut ajoympäristön ominaisuustiedot. Palvelimen tulee kyetä osoittamaan testitapaukset sopiville ympäristöille.

Tämän lisäksi palvelimen tulee kyetä kokoamaan testitapauksista yhdistelmiä, jotka vastaavat kestoltaan mahdollisimman tarkasti annettua aikaa. Testiautomaatioajojen kesto määräytyy siten, että arkipäivisin ne ovat noin 10 tunnin mittaisia. Viikonloppuajot voivat kestää jopa useamman vuorokauden ajan.

Palvelimella tulee olla logiikkaa, jolla testitapausten priorisointia voidaan toteuttaa. Priorisointiin vaikuttavat keston ja ajohistorian lisäksi prioriteetti merkinnät. Prioriteetiltaan vähäisiksi merkittyjä testitapauksia ajetaan pääsääntöisesti vain viikonloppuisin. Palvelimen tulee osata priorisoida pakolliset ja välttämättömät testitapaukset ajettavaksi jokaiselle testausympäristölle. Tällaisia pakollisia ja suuren prioriteetin testitapauksia ovat esimerkiksi kalibrointitestit.

Opinnäytetyössä toteutettavalle palvelimelle asetetaan suorituskykyyn liittyviä vaatimuksia, jotka liittyvät muun muassa vasteaikaan. Vasteajalla tarkoitetaan aikaa, joka palvelimella kestää toteuttaa haluttu toimenpide. Vasteaika mitataan pyynnön saapumisesta alkaen. Palvelimen tulee palauttaa testiympäristölle osoitetut testitapaukset kohtuullisessa ajassa, jonka voidaan ajatella olevan maksimissaan 1 minuutin. Vasteaikaa tärkeämpää palvelimelle on varmatoimisuus ja palvelimen stabiilius. Mahdolliset ongelmat palvelimessa tai sen toimimattomuus aiheuttaa merkittävän ongelman koko testiautomaatioprosessille.

Näiden vaatimusten lisäksi palvelinohjelmistolle voidaan asettaa implisiittisiä vaatimuksia. Implisiittisillä vaatimuksilla tässä tapauksessa tarkoitetaan esimerkiksi dokumentaatiota ja laajennettavuutta useampaan ympäristöön tulevaisuudessa. Implisiittiset vaatimukset ovat ominaisuuksia, jotka voidaan ajatella olevan yleisiä. Ne eivät kuulu toiminnallisiin tai ei-toiminnallisiin vaatimuksiin.

6.2 Tekninen suunnittelu

Tässä kappaleessa käsitellään palvelimen toteutuksessa käytettäviä tekniikoita ja teknisiä ratkaisuja. Palvelin toteutetaan C# (CSharp) ohjelmointikielellä. C# on yleinen ohjelmointikieli ja sille on useita ohjelmistokehyksiä, joilla voidaan toteuttaa erilaisia sovellus – ja palvelinratkaisuja. Palvelimen toteutuksessa käytetään ASP .NET – ohjelmistokehystä. Se on yleisesti tunnettu ja laajasti käytetty ohjelmistokehys verkkosovelluksille.

ASP. NET on Microsoftin julkaisema avoimen lähdekoodin ohjelmistokehys. Se antaa kehittäjille mahdollisuuden käyttää C++ tai C# ohjelmointikieliä. Ominaisuudet, jotka tekevät siitä suositun ovat mm. tiedostopohjainen ohjaustapa ja sisäänrakennetut tietoturvaominaisuudet. Lisäksi se mahdollistaa selkeän ja helposti ylläpidettävän ohjelmistokoodin. (Yuhiro Global, 2023)

Palvelimen toiminnan kannalta on tärkeää tietää testitapausten oleelliset tiedot. Ponsse Oyj:n testiautomaatiojärjestelmässä näitä tietoja on tallennettu Microsoft SQL – tietokantaan. Tämän tietokannan hyödyntäminen palvelimella osoittautui alustavassa suunnittelussa haasteelliseksi. Tietokannassa oleva tieto oli puutteellista kokonaisten testisarjojen osalta.

Testitapausten ajohistorian tallentaminen toteutetaan paikalliselle SQLite – tietokannalle. Tietokantajärjestelmän valinnassa päädyttiin SQLite:n käyttöön, koska se on kevyt ja helposti hallittava paikallisesti toimiva tietokanta. SQLiten lisäksi harkittiin muitakin tietokantaratkaisuja ja tiedon tallentamisen muotoja. Palvelimen tarpeet huomioiden se oli optimaalisin ratkaisu tähän tarpeeseen.

”SQLite on C – Ohjelmointikielinen kirjasto, joka toteuttaa pienen, nopean, itsenäisen, erittäin luotettavan ja monipuolisen tietokantamoottorin” (SQLite, ei pvm) Sen etuja ovat erityisesti keveys ja helppokäyttöisyys. Tietokannan sisältö voidaan tallentaa yhteen tiedostoon, joka tekee siitä hyvän valinnan pienen tietokantaa tarvitsevan sovelluksen osaksi. (Laaksonen, ei pvm) SQLite on hyvä ratkaisu silloin, kun tietomäärä on vähäistä. Datan kasvaessa voi olla perusteltua harkita toisenlaisia tietokantaratkaisuja.

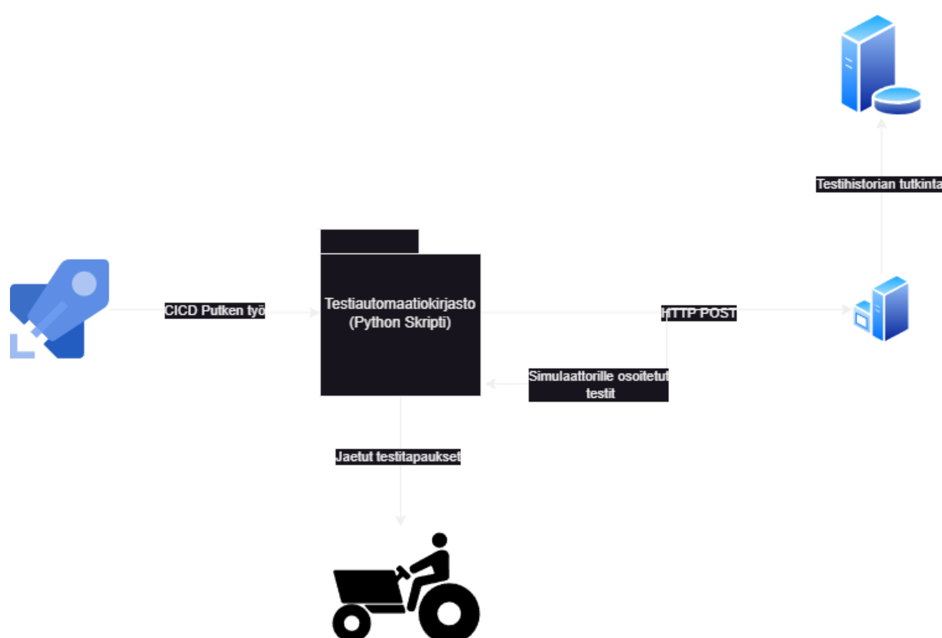
Näiden lisäksi palvelimen toteutuksessa tullaan käyttämään yleisesti tunnettuja ja luotettavia ohjelmistopaketteja, jotka helpottavat ja nopeuttavat ohjelmointia. Ohjelmistopaketit eli niin kutsutut NuGet – paketit ovat ladattavia ohjelmiston osia, joilla voidaan toteuttaa yleisiä ohjelmistoon liittyviä toiminnallisuuksia. Palvelimella tullaan käyttämään NewtonSoftin Json – pakettia, jolla hoidetaan Json – formaatin mukaisen tiedon muuntaminen.

Opinnäytetyössä toteutettava palvelin otetaan käyttöön Ponsse Oyj:n sisäverkossa, jolloin se on käytettävissä testiautomaatiossa. Palvelimen ajoympäristönä tulee toimimaan Windows – palvelin.

Alla olevassa kuvaajassa kuvataan palvelimen toiminta osana testiautomaatioympäristöä. Palvelimelle lähetetään http – pyyntöjä Python skriptillä, joka on osana testiautomaatiokirjastoa. Pyynnössä annetaan tarvittavat tiedot palvelimelle testitapausten kokoamiseen. Tällaisia tietoja ovat:

- Testiajon pituus
- Ohjelmistokonfiguraatiot
- Testausympäristön fyysiset lisälaitteet

Pyyntöjen lähettäminen tullaan hoitamaan olemassa olevan CI/CD – putken kautta. Testitapaus historia tietokannan ylläpito tapahtuu ohjelmallisesti ja ajastetusti palvelimella. Alla olevassa kuvassa esitellään testiautomaatioprosessia, jonka osaksi palvelin integroituu (Kuva 13).



Kuva 13. Palvelin osana testiautomaatioprosessia (Viljamaa, 2024)

7 TOTEUTUS

7.1 Tietokanta

Palvelin tarvitsee toimiakseen ajantasaiset tiedot jaettavista testisarjoista, sekä aiemmista testituloksista. Ponssen testiautomaatiojärjestelmä käyttää Robot Frameworkilla luotuja testitapauksia, jotka ovat saatavilla Ponssen sisäisestä versionhallintaympäristöstä. Koska uusia testejä tai muutoksia jo olemassa oleviin testeihin tehdään usein, on tärkeää, että palvelimen käytössä on aina uusimmat versiot jaettavista testitapauksista. Samoin tiedot aiemmista testituloksista on pidettävä ajan tasalla palvelimen toiminnan takaamiseksi. Näiden ominaisuuksien toteutukseen palvelin hyödyntää Hangfire-kirjastoa, jonka avulla voidaan luoda ja hallinnoida taustalla ajettavia tehtäviä. Kun palvelin käynnistetään, se luo uuden ajastetun tehtävän, joka suorittaa sekä testitapausten, että aiempien testitulosten päivittämisen tietokantaan. Koska käyttäjämäärät ovat pieniä ja käyttö on paikallisesti rajattua, tietokannaksi valittiin SQLite. Tietokannassa on kaksi taulua, yksi jaettaville testeille ja toinen aiempia testituloksia varten.

7.1.1 Jaettavien testien ylläpito

Uusimmat testitapaukset haetaan ensin versionhallinnasta .bat-skriptin avulla. Tämän jälkeen palvelin hyödyntää Python-skriptiä, jonka tehtävänä on suodattaa ja valita jaettavaksi tarkoitetut testit. Testit, jotka on merkattu esimerkiksi development-tagilla, rajataan pois. Nämä testit on tarkoitettu ajettavaksi erillisissä ajoissa, eikä niitä tule ottaa jaettavaksi normaaleihin testiajoihin. Vaikka palvelin on muilta osin toteutettu C#-ohjelmointikielellä, testitapausten luku toteutettiin Pythonilla. Robot Framework -kirjasto on toteutettu Pythonilla, joten skripti voi hyödyntää suoraan kirjaston tarjoamia rajapintoja testitapausten käsittelyyn. Testien läpikäynnissä hyödynnettiin Robot Frameworkin Visitor-rajapintaa, joka tarjoaa strukturoidun tavan käsitellä testitapauksia ja niihin liittyviä tietoja. Käyttämällä tätä rajapintaa skripti keräsi jokaiselle testitapaukselle:

- Nimen: testisarjan yksilöllinen nimi
- Relatiivisen polun: suhteellinen polku testitapauskansion juuresta testisarjan sijaintiin
- Tagit: testisarjaan liitetyt tagit, jotka kuvaavat sen ominaisuuksia tai käyttötarkoitusta

Tämän jälkeen skripti analysoi testisarjan tageja, ja jos ne olivat sopivia tuotantoympäristön ajoihin, se lisättiin jaettavien testien listaan. Viimeisenä tämä lista kirjoitettiin json-muotoisena levylle, josta palvelin lukee tiedot, ja tallentaa ne tietokantaan (Kuva 14).

Suites	
PK	Id
Text	Name
Text	Path
Text	Tags

Kuva 14 Testisarjat-taulu tietokannassa.

7.1.2 Testihistorian ylläpito

Palvelin kerää ja tallentaa tietoa aiemmista testiajoista. Robot Framework tuottaa testiajoista useita lokitiedostoja. Testihistorian ylläpitoon käytännöllisin vaihtoehto oli XML-muotoinen lokitiedosto, joista parsittiin seuraavat tiedot jokaisesta testisarjasta:

- Testisarjan nimi: suoritetun testisarjan nimi
- Kesto: aikamäärä, joka kului testisarjan suorittamiseen
- Konemalli ja lisälaitteet: konemalli ja sen käyttämät laitteet, kuten nosturin tyyppi ja käytetyt kahvat
- Ajon suorittanut ympäristö: onko testiajo suoritettu fyysisellä laitteistolla vai simuloituna
- Virheiden määrä: monta virhettä suorituksen aikana ilmeni
  Aikaleima: ajankohta, jolloin testisarja suoritettiin

Tiedot luetaan ja tallennetaan tietokantaan testisarjojen tulostauluun (Kuva 15), jotta niitä voidaan käyttää myöhemmin testienjakovaiheessa.

SuiteResults	
PK	Id
Text	Name
Text	Duration
Text	Machine
Text	Environment
Int	Failures
DateTime	Timestamp

Kuva 15 Aiempien testiajojen taulu tietokannassa.

7.2 Palvelimen toiminta

Koska C#-ohjelmointikieltä ja .NET Framework 4.8 -ympäristöä käytetään laajalti yrityksessä, päätettiin palvelin toteuttaa niiden avulla. Web-palvelimen toteutuksessa käytettiin ASP.NET Web API -sovelluskehystä. ASP.NET tarjoaa yksinkertaisen tavan luoda rajapintoja ilman aikaa vievää kehitystyötä. Tämän ansiosta voidaan keskittyä projektin kannalta olennaiseen tekemiseen, eli testienjakopalvelun kehittämiseen ja asiakastarpeiden täyttämiseen ilman ylimääräistä vaivaa oman web-palvelin-toteutuksen kehittämisestä ja ylläpitämisestä.

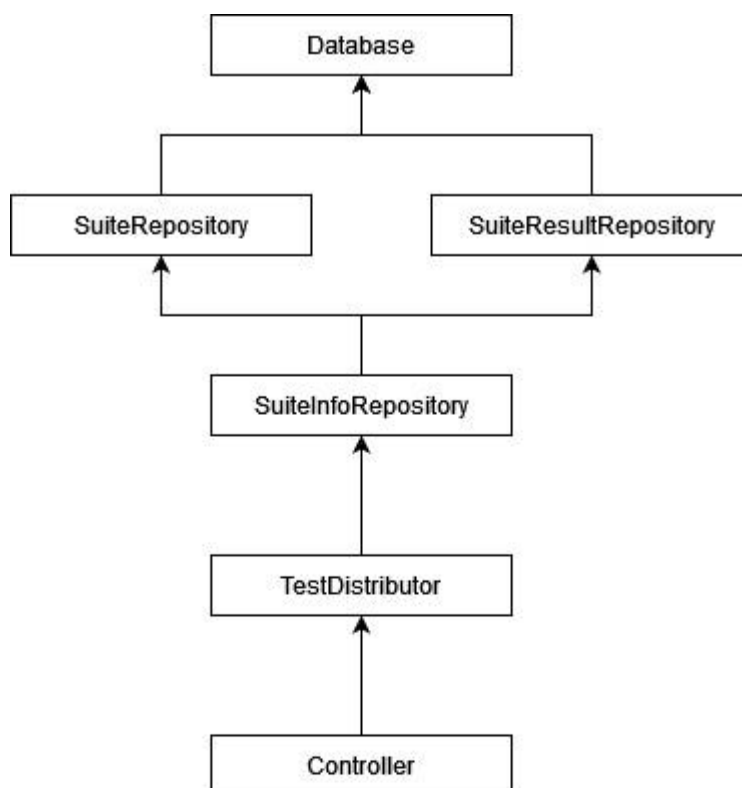
Palvelimen tarjoamalla rajapinnalla on kolme reittiä. Kaksi ensimmäistä reittiä ovat tietokannan manuaaliseen päivitykseen, eli niiden kautta voidaan käynnistää testitapausten tai testihistorian päivitys tietokantaan. Tämä ominaisuus oli käytössä etenkin kehityksen aikana, mutta se jätettiin myös lopulliseen versioon, jotta tietokanta voidaan päivittää tarvittaessa ilman palvelimen uudelleenkäynnistystä.

Viimeinen reitti toteuttaa palvelimen pääasiallisen tehtävän, se siis palauttaa vastauksena Robot Frameworkia varten argumenttitiedoston, joka sisältää ajettavat testisarjat. Testiympäristön lähettämän

HTTP-pyynnön body-osio sisältää JSON-objektin, joka tarjoaa tietoa pyynnön lähettäneestä ympäristöstä, kuten esimerkiksi testiajojen halutun keston ja ajoissa käytettävän konemallin sekä koneessa.

Yksittäisten testien sijaan järjestelmä jakaa kokonaisia .robot-tiedostoja, joita kutsutaan myös testisarjoiksi. Normaalisti testisarjan valmisteluvaihe (engl. suite setup) suoritetaan vain kerran koko testisarjan alussa. Jos testejä jaettaisiin yksittäisinä testitapauksina, testisarjan aloittava valmisteluvaihe toistuisi jokaisen testin kohdalla, mikä lisäisi suoritusaikaa merkittävästi. Vaikka testienjakojärjestelmä keskittyykin testisarjojen jakamiseen, yksittäisten testien jakaminen olisi mahdollista. Jakamalla yksittäisiä testejä, olisi teoriassa mahdollista vaikuttaa testiajon kestoon vieläkin suuremmalla tarkkuudella, sillä yksittäisen testin kesto on lähes poikkeuksetta lyhyempi, kuin kokonaisen testisarjan vaatima aika. Käytännössä tämä lähestymistapa kuitenkin lisäisi testiajojen suoritusaikaa edellä kuvatun valmisteluvaiheen monistumisen vuoksi.

7.2.1 Arkkitehtuurin yleiskuvaus



Kuva 16 Yksinkertaistettu versio palvelimen arkkitehtuurista.

Palvelimen toiminta perustuu muutamaan pääkomponenttiin (Kuva 16):

- Database-luokka huolehtii tietokantaan yhdistämisestä ja se tarjoaa rajapinnan SQL-kyselyiden suorittamiseen. Jos tietokantaa tai tarvittavia tauluja ei ole vielä olemassa ohjelman käynnistyessä, ne luodaan automaattisesti.
- Suite- ja SuiteResultRepository -luokat tarjoavat rajapinnan testisarjojen ja testihistorian lukemiseen ja tallentamiseen tietokantaan. Tietokannasta haetut tiedot luetaan tietokantatauluja vastaaviin C#-luokkiin, jotta niiden jatkokäsittely olisi helpompaa.

- SuiteInfoRepository vuorostaan yhdistää testisarjat ja niihin liittyvät historiatiedot yhden luokan alle. Näin itse testien jakamisesta vastaavalla luokalla on käytössä tarvittavat tiedot testien jakoa ja priorisointia varten.
- TestDistributor-luokan vastuulla on ylläpitää listaa jaettavista testeistä, sekä jakaa sopivat testit ympäristöille prioriteetin ja pyynnön mukana tulleen konfiguraation perusteella.
- Kontrollerin tehtävänä on vastaanottaa tulevat pyynnöt, tarkistaa välittää pyynnön mukana olleet tiedot TestDistibutor-luokalle ja lopulta palauttaa valmis testitiedosto pyynnön tehneelle ympäristölle.

```
 1 [HttpPost]
 2 [Route("suites")]
 3 IHttpActionResult GetSuites(MachineConfiguration conf)
 4 {
 5   if (!ModelState.IsValid) {
 6     return BadRequest(ModelState);
 7   }
 8
 9   DistributedSuites suites = _suiteDistributor.GetSuites(conf);
10   return Ok(suites);
11 }
```

Kuva 17 Testit palauttava reitti.

Kontrolleri kuuntelee HTTP-protokollan mukaisia POST-metodilla lähetettyjä pyyntöjä ”suites”-reittiin (Kuva 17). Pyynnön body-lohkossa odotetaan olevan ympäristön konfiguraatiota vastaava json-objekti (Kuva 18). ASP.NET osaa muuntaa tämän json-objektin suoraan C#-luokaksi, eli tässä tapauksessa ”MachineConfiguration”-luokan ilmentymäksi. Kontrolleri myös tarkistaa, että pyynnön mukana tullut konfiguraatio sisältää kaikki tarvittavat tiedot. Virhetilanteessa palvelin palauttaa virheellisestä konfiguraatiosta kertovan viestin.

```
 1 {
 2   "Duration": 8,
 3   "Machine Type: "Scorpion King",
 4   "Crane": "C5
 5       ...
 6   "Specialties": {
 7     "ConnectivityUnit": true,
 8     "IsHIL": true
 9   }
10 }
```

Kuva 18 Esimerkki ympäristön lähettämästä konfiguraatiosta.

Palvelin tarvitsee testien jakoa varten seuraavia tietoja testejä suorittavalta ympäristöltä:

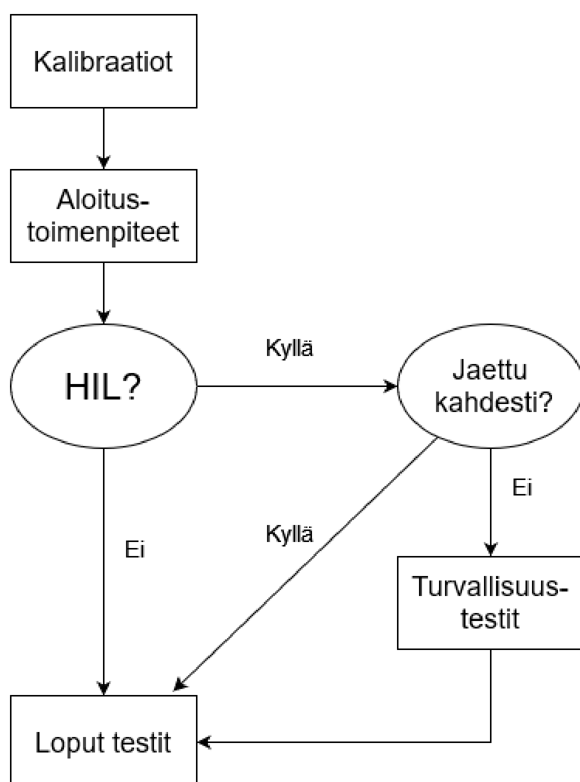
1. Testiajon haluttu kesto
2. Konetyyppi, jolla testit suoritetaan
3. Testejä suorittavan koneen eri osien tyypit, esimerkiksi nosturin ja kahvojen mallit
4. Erikoislaitteisto, joita ympäristössä on. Esimerkiksi tiedonsiirtotestit vaativat toimiakseen ympäristöltä tiettyjä ominaisuuksia.
5. Onko ympäristö SIL- vai HIL-tyyppinen

Jos ympäristö toimitti tarvittavat tiedot, palvelin ohjaa pyynnön sisältämät tiedot testien jaosta vastaavalle luokalle.

7.3 Testien jako

Palvelimen käynnistyessä TestDistributor-luokasta luodaan yksi instanssi, ja se annetaan riippuvuusinjektioista huolehtivalle Unity Container -kirjaston käyttöön. Unity Container konfiguroitiin palauttamaan luokka singleton-tyyppisenä, eli yhtä ja samaa instanssia käytetään koko palvelimen käynnissä olon ajan. Koska testien jaosta vastaavasta luokka ylläpitää myös listaa jaettavista testeistä, ja jo jaetut testit poistetaan tästä listasta, tulee instanssin tilan säilyä myös pyyntöjen välissä.

Koneen kalibraatioon ja aloitustoimenpiteisiin liittyvät testit on eritelty omiin listoihinsa. Nämä testit ajetaan jokaisella ympäristöllä poikkeuksetta, eikä niitä haluta poistaa jaettavista testeistä missään vaiheessa. Palvelin lisääkin heti ensimmäisenä konekonfiguraation perusteella ympäristön tarvitsemat kalibraatiotestit, sekä alkutoimenpiteitä suorittavia testejä, jotka sisältävät esimerkiksi myöhemmissä testeissä tarvittavien käyttäjien luonnin. Nämä testit eivät siis ole priorisoituja saman tyyppisesti, kuin jäljelle jäävät testit, vaan ne käsitellään erikoistapauksina. (Kuva 19)



Kuva 19 Testisarjojen jakologiikka kaaviona.

7.4 Testien priorisointi

Seuraavaksi palvelin tarkistaa, onko ympäristö HIL- vai SIL-tyyppinen. Jos kyseessä on HIL-ympäristö, korkeimman prioriteetin testisarjoiksi nousevat turvallisuusominaisuuksia testaavat testisarjat. Nämä testit testaavat nimensä mukaisesti turvallisuuteen liittyviä ominaisuuksia, kuten järjestelmän kykyä havaita ja reagoida virheisiin ja vaaratilanteisiin. Tavoitteena on varmistaa, että järjestelmä toimii turvallisesti kaikissa olosuhteissa ja suojaa käyttäjäänsä mahdollisilta vaaratilanteilta. Tämän vuoksi on erityisen tärkeää, että nämä testit jaetaan suurimmalla mahdollisella prioriteetilla. Kun turvallisuustestejä on saatu jaettua kahdelle eri HIL-ympäristölle, nämä testit voidaan poistaa jaettavien testien listasta.

Viimeisenä palvelin hakee pyynnön tehneelle ympäristölle muita sopivia testisarjoja, kunnes haluttu aikaraja on tullut täyteen. Yksittäisten testisarjojen kestoa arvioidaan laskemalla mediaani muutaman aiemman tuloksen perusteella. Tämä antaa kohtuullisen tarkan arvion testisarjan kestosta ja ottaa huomioon myös mahdolliset poikkeavuudet yksittäisissä suorituksissa. Pidemmällä aikavälillä se myös mukautuu testien muutoksiin, jotka voivat vaikuttaa testisarjan kestoon.

Nämä jäljelle jääneet testisarjat on priorisoitu kahden kriteerin perusteella:

1. Viime ajossa epäonnistuneet testisarjat
2. Testisarjat, joiden viimeisimmästä ajosta on pisin aika

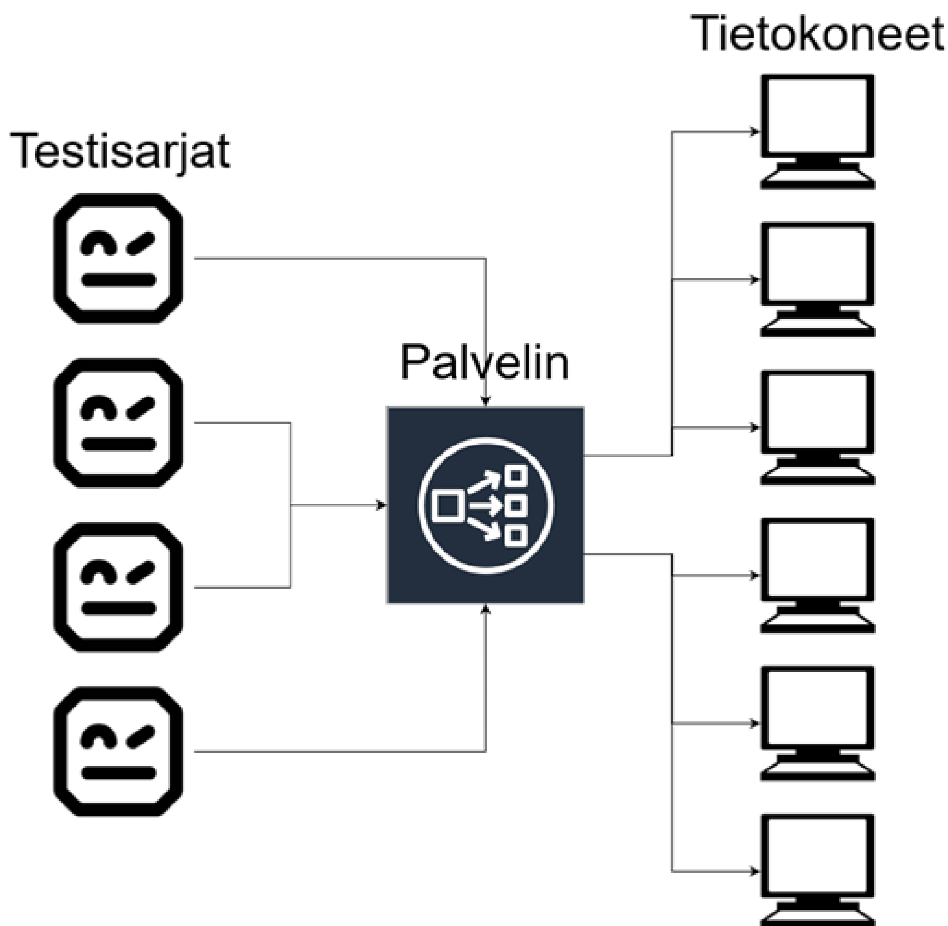
Ympäristölle jaetut testisarjat poistetaan tämän jälkeen jaettavien testien listasta, jotta niitä ei suoriteta useaan kertaan yhden ajon aikana. Jos jostain syystä ympäristölle jaettujen testien kesto on alle 80 % tavoitekestosta, TestDistributor-luokan tila nollataan, ja ympäristölle yritetään lisätä testisarjoja, kunnes tavoite on saavutettu. Jaettavien testien lista nollataan myös ajastetusti kerran päivässä, joten jos aiemmin mainittu tilanne havaitaan, se logitetaan, jotta syytä voidaan yrittää selvittää.

Priorisointistrategian tavoitteena on tehostaa testauksen laatua ja tehokkuutta. Turvallisuusominaisuuksien jälkeen viime ajoissa epäonnistuneet testisarjat saavat etusijan, koska niiden epäonnistuminen viittaa ongelmiin testattavassa ohjelmassa. Jos testisarja on epäonnistunut edellisenä yönä, on todennäköisempää, että se epäonnistuu myös seuraavissa ajoissa, verrattuna testisarjaan, joka suoriutui virheettömästi. Viimeisenä palvelin jakaa testisarjoja ympäristölle testisarjojen viimeisimmän ajopäivämäärän perusteella. Mitä kauemmin aikaa testisarjan ajamisesta on kulunut, sitä nopeammin se halutaan ajettavaksi. Näin pidemmällä aikavälillä jokainen testisarja tulee ajetuksi säännöllisesti. Myös uudet testisarjat, joita ei ole vielä suoritettu kertaakaan, saavat suuren prioriteetin, sillä tässä erikoistapauksessa viimeisintä ajoaikaa ei ole, joten sille on asetettu oletusarvo, joka varmistaa, että ne ajetaan mahdollisimman pian.

7.5 Kuormanjako palvelimen näkökulmasta

Projektissa jaettava kuormana toimivat testiautomaation testisarjat, joita on noin 500 kappaletta (Kuva 20). Testisarjojen testit on ryhmitelty kokonaisuuksiin, eli testit, jotka liittyvät toisiinsa toimin-

nallisesti on koottu yhteen testisarjaan - esimerkiksi nosturin toimintaa ja tiedonsiirtoa testaavat testit ovat omissa testisarjoissaan. Kuormanjakajana toimiva palvelin jakaa testeistä halutun mittaisia kestäviä testiajoja, jotka testiautomaation käytössä olevat ympäristöt suorittavat.



Kuva 20 Testisarjat toimivat jaettavana kuormana.

Palvelimen käytössä oleva kuormanjaon algoritmi on luonteeltaan dynaaminen, sillä palvelimen tila vaikuttaa siihen, mitä testejä ympäristöille jaetaan. Aiempien pyyntöjen perusteella jaetut testisarjat poistuvat jaettavien testien listasta, joten saapuvien pyyntöjen järjestys vaikuttaa myös testisarjoihin, joita pyynnön tehneelle ympäristölle jaetaan. Koska saapuvien pyyntöjen järjestys ei ole etukäteen tiedossa ja palvelimen haluttiin kykenevän käsittelemään minkälaisia konekonfiguraatioita tahansa, koko kuormanjakoprosessi tapahtuu reaaliajassa. Lisäksi on mahdollista, että jokin ympäristö on tilapäisesti poissa käytöstä öisistä testiajoista. Näin ollen palvelin ei voi suorittaa jakoa etukäteen, sillä olisi mahdollista, että esimerkiksi turvallisuustestit olisi jaettu juuri sille ympäristölle, joka on pois käytöstä.

8 POHDINTA JA JOHTOPÄÄTÖKSET

Tämän opinnäytetyön tarkoituksena oli tuottaa lisäarvoa Ponsse Oyj:n testiautomaatiolle ja tehostaa testausautomaatioprosessia. Suuren testimassan ja rajallisen testauskapasiteetin aiheuttamaa ongelmaa lähdettiin ratkaisemaan priorisoinnilla. Testitapausten priorisointi osoittautui sopivaksi lähestymistavaksi alkuperäiseen ongelmaan testausprosessin ollessa jo nykyisellään suunniteltu toimimaan tehokkaasti. Priorisointi valittiin lähestymistavaksi, koska se nähtiin ainoaksi tavaksi tuottaa lisäarvoa toimeksiantajan testiautomaation järjestämisessä. Testausprosessin tehostamisella ei olisi saatu yhtä kokonaisvaltaisia vaikutuksia. Palvelimen rakentaminen mahdollistaa ohjelmiston laadunvarmistuksen parhaan mahdollisen toteuttamisen ilman suuria panostuksia, kuten uusiin testausympäristöihin investoinnit.

Testitapausten priorisoinnissa päädyttiin palvelimeen, jolle määriteltiin priorisointiin vaikuttavat tekijät. Priorisointia ja sen perusteita käytiin läpi raportin kappaleessa 7. Opinnäytetyön ongelman ratkaisuna palvelin todettiin parhaana mahdollisena lähestymistapana. Sen valintaan päädyttiin, koska opinnäytetyön ongelmaan ei löydetty toimivaa ja olemassa olevaa ratkaisua. Priorisoinnin lisäksi opinnäytetyössä toteutettavan palvelimella haluttiin vähentää manuaalista työtä testitapausten määrittelyssä eri testausympäristöille. Tämä otettiin huomioon palvelimen toteutuksessa. Palvelin tulee vaatimaan jatkokehitystä ja ylläpitoa, joten manuaalisesta työstä ei tälläkään ratkaisulla päästä täysin eroon. Palvelin kuitenkin vähentää manuaalista työtä merkittävästi ja palvelimen kehittyessä tämän työn määrä vähenee entisestään.

Tämän tutkimus– ja kehittämistyön tuloksena saatiin ratkaisu suuren testimassan aiheuttamaan ongelmaan toimeksiantajan testiautomaatiossa. Opinnäytetyössä päästiin tavoitteeseen, vaikka palvelinta ei saatettu tuotantokäyttöön opinnäytetyön puitteissa. Ennen palvelimen tuotantokäyttöön saattamista vaaditaan koekäyttöä ja testaamista, jotta voidaan varmistaa sen haluttu toiminta.

Opinnäytetyön aikana palvelinta on testattu paikallisesti. Paikallisesti toteutetut testit kuitenkin vaikuttavat lupaavilta – palvelin jakaa testejä halutuilla prioriteeteilla ja osaa ottaa huomioon myös ympäristöille asetetut erikoisominaisuudet. Palvelimen saattaminen osaksi testiautomaatiota ja testausputkea vaatii koekäyttöä oikeassa testausympäristössä. Tämä tarkoittaa palvelimen ajamista halutussa palvelinympäristössä ja testien hakemista CICD –putkiston kautta. Näin varmistetaan palvelimen toiminta ja havaitaan mahdolliset poikkeamat, jotka voivat aiheuttaa ongelmia testisarjojen määrittelyssä. Tämä testaus ja käyttöönotto tullaan toteuttamaan opinnäytetyön jälkeisenä aikana töiden jatkuessa toimeksiantajalla.

Opinnäytetyön tavoitteena oli toteuttaa ratkaisu ongelmaan, vaikka realistisesti ajateltuna ratkaisu tulee vaatimaan vielä työpanosta jatkokehityksen osalta. Toteutettu palvelin on todettu toimivaksi ratkaisuksi ongelmaan, mutta täysin varmatoimista ja lopullista ratkaisua ei voitu toteuttaa tämän opinnäytetyön puitteissa. Palvelimen toimintaan liittyvät jatkokehitystarpeet tulevat todennäköisesti ilmi pidemmällä testausjaksolla. Jatkokehitystä ja palvelimen toimintaa voidaan joutua jatkossa laajentamaan esimerkiksi uusien konemallien tai uusien ominaisuuksien mukaan.

Työssä on käytetty seuraavasti tekoälyä:

ChatGPT 2024. OpenAI. GPT-3.5. Käytetty kielentarkistukseen, helmi-toukokuu 2024. https://chatgpt.com/

ChatGPT 2024. OpenAI. GPT-4o. Käytetty kielentarkistukseen, toukokuu 2024. https://chatgpt.com/

LÄHTEET

Alakeel, A. (2009). A Guide to Dynamic Load Balancing in Distributed Computer Systems. *International Journal of Computer Science and Network Security, 10*(6).

Amazon. (ei pvm). *What is devops?* Haettu 22.3.2024 osoitteesta https://aws.amazon.com/devops/what-is-devops/

AnAr Corporate. (15.2.2023). *Programming languages for automation testing*. Haettu 23.2.2024 osoitteesta https://anarsolutions.com/programming-languages-for-automation-testing/

Aniche, M. (2022). *Effective Software Testing: A developer's guide.* Shelter Island: Manning Publications Co.

Appium. (2024). *Appium in a Nutshell*. Haettu 25.2.2024 osoitteesta https://appium.io/docs/en/latest/intro/

Bierig, R.;Brown, S.;Galván, E.;& Timoney, J. (2022). *Essentials of Software Testing.* Cambridge: Cambridge University Press.

Boehm, B. W.;& Papaccio, P. N. (1988). Understanding and Controlling Software Costs. IEEE Transactions on Software Engineering.

Chauran, N. (2010). *Software testing - principles and practices.* Oxford University Press.

Chengzhong, X.;& Francis, C. M. (1997). *Load Balancing in Parallel Computer - Theory and Practice.* Norwell: Kluwer Academic Publishers.

Chickowski, E. (ei pvm). *5 red flags: When DevOps might not be a good fit*. Haettu 14.4.2024 osoitteesta https://techbeacon.com/app-dev-testing/5-red-flags-when-devops-might-not-be-good-fit

Cohn, M. (2010). *Succeeding With Agile - Software development using Scrum.* Ann Arbor: Addison-Wesley.

Dawson, M.;Burrell, D.;Rahim, E.;& Stephen, B. (2010). Integrating Software Assurance into the Software Development Life Cycle. *Journal of Information Systems Technology and Planning*.

Dijkstra, E. W. (1970). *Notes on structured programming.* Eindhoven: Techonological University Eindhoven.

Doshi, K. (22.3.2023). *BrowserStack*. Haettu 16.2.2024 osoitteesta https://www.browserstack.com/guide/types-of-testing

Dunster, B. (ei pvm). *How does devops compare to traditional development methods?* Haettu 14.4.2024 osoitteesta https://www.clouddirect.net/how-does-devops-compare-to-traditional-development-methods/

GitLab. (ei pvm). *DevOps tool explained*. Haettu 2.4.2024 osoitteesta https://about.gitlab.com/topics/devops/devops-tools-explained/

Goodenough, J.;& Gerhart, S. (1.4.1975). Toward a theory of test data selection. *ACM SIGPLAN Notices - kuukausilehti*(10), ss. 493 - 510. Haettu 22.2.2024

Hamilton, T. (23.3.2024). *Guru99*. Haettu 17.4.2024 osoitteesta Mikä on CI/CD? Jatkuva integrointi ja jatkuva toimitus: https://www.guru99.com/fi/continuous-integration.html

IteWiki. (ei pvm). *DevOps*. Haettu 8.4.2024 osoitteesta https://www.itewiki.fi/opas/devops/

Jose, B. (2021). *Test automation: A Manager's Guide.* Swindon: BCS Learning & Development Limited.

JumpGrowth. (2023). *What is the future of DevOps?* Haettu 24.3.2024 osoitteesta https://jumpgrowth.com/what-is-the-future-of-devops/

Kariappa, P. (24.2.2024). *12 Business Benefits of CI/CD*. Haettu 17.3.2024 osoitteesta https://www.opsera.io/blog/ci-cd-business-benefits

Laaksonen, A. (ei pvm). *SQLite vs. PostgreSQL*. Haettu 6.4.2024 osoitteesta https://hy-tsoha.github.io/materiaali/sqlite_postgre/

LambdaTest. (28.2.2023). Manual Testing Tutorial. Haettu 20.2.2024 osoitteesta https://www.lambdatest.com/learning-hub/manual-testing

Magowan, K. (julkaisuaika tuntematon). Shift Left Testing: What, Why & How To Shift Left. Haettu 25.3.2024 osoitteesta https://www.bmc.com/blogs/what-is-shift-left-shift-left-testing-explained

Metatavu Oy. (ei pvm). *Vaatimusmäärittely: Mitä se tarkoittaa ja miksi se on tärkeä osa projektin onnistumista?* Haettu 30.3.2024 osoitteesta https://metatavu.fi/vaatimusmaarittely-mita-se-tarkoittaa-ja-miksi-se-on-tarkea-osa-projektin-onnistumista/

Mosse, D. (julkaisuaika tuntematon). Load Balancing. Haettu 15.4.2024 osoitteesta http://www.cs.pitt.edu/%7Emosse/cs2510/class-notes/load-balancing.pdf

Nabil, M. (ei pvm). *Android Test Automation with Espresso*. Haettu 19.3.2024 osoitteesta https://testautomationu.applitools.com/espresso-mobile-testing-tutorial/

Padilha, L. (2018). Analysis of the use of SDN for load balancing. University of Sao Paulo.

Patton, R. (2001). *Software Testing.* Sams Publishing.

Ponsse Oyj. (julkaisuaika tuntematon). Opti 5G - Metsien monipuolisin tietojärjestelmä. Haettu 6.5.2024 osoitteesta https://pim.ponsse.com/media/ponsse-pim-api/api/content/getfile/17045277.pdf

Robot Framework. (2024). *Getting started*. Haettu 2.5.2024 osoitteesta https://robotframework.org/#getting-started

SmartBear Software. (ei pvm). *Test Automation Frameworks*. Haettu 19.2.2024 osoitteesta https://smartbear.com/learn/automated-testing/test-automation-frameworks/

Son, H. (27.12.2023). Agile Testing Methodology: Life Cycle, Techniques, & Strategy. Haettu 1.5.2024 osoitteesta https://www.testrail.com/blog/agile-testing-methodology/

SQLite. (ei pvm). *What is SQLite?* Haettu 6.4.2024 osoitteesta https://www.sqlite.org/

Swartout, P. (2018). *Continuous Delivery and DevOps - a Quickstart Guide.* Packt Publishing, Limited.

Synopsys. (ei pvm). *What is devops?* Haettu 25.3.2024 osoitteesta https://www.synopsys.com/glossary/what-is-devops.html

Taina, J. (2005). *Ohjelmistotuotanto, vaatimusanalyysi*. Haettu 1.3.2024 osoitteesta https://www.cs.helsinki.fi/u/taina/ohtu/k-2005/2luku4.pdf

Tieturi Oy. (23.11.2023). *8 Syytä opiskella Javaa vielä vuonna 2024*. Haettu 11.3.2024 osoitteesta https://www.tieturi.fi/blogi/8-syyta-opiskella-javaa-viela-vuonna-2024/

Umbraco. (julkaisuaika tuntematon). https://umbraco.com/knowledge-base/load-balancing/. *What is load balancing?* Haettu 22.2.2024 osoitteesta https://umbraco.com/knowledge-base/load-balancing/

VALA Group. (2021). *Testiautomaatio-opas*. Haettu 16.2.2024 osoitteesta https://www.valagroup.com/wp-content/uploads/2021/12/testiautomaatio-opas-2020-min.pdf

VALA Group. (24.10.2023). Integraatiotestaus: Mitä se on? *Ohjelmistotestaus blogi*. Haettu 19.2.2024 osoitteesta https://www.valagroup.com/fi/blogi/integraatiotestaus/

Vala Group. (30.1.2023). *Vala Group*. Haettu 22.2.2024 osoitteesta https://www.valagroup.com/fi/blogi/opas-testaussuunnitelman-laatimiseen-ja-ilmainen-malli/

Yuhiro Global. (6.6.2023). *ASP.NET MVC vs. ASP.NET Core: Mitä käyttää?* Haettu 2.5.2024 osoitteesta https://www.yuhiro-global.com/fi/asp-net-mvc-vs-asp-net-core-mita-kayttaa/