TIETO- JA SÄHKÖTEKNIIKAN TIEDEKUNTA

Antti Keränen

Automaatiotestaaminen web-sovelluksen kehityksessä

Diplomityö
Tietotekniikan tutkinto-ohjelma
Huhtikuu 2023

Keränen A. (2023) Automaatiotestaaminen web-sovelluksen kehityksessä. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 50 s.

TIIVISTELMÄ

Digitalisoituminen ohjaa finanssialan sovellusten kehitysmenetelmiä kohti ketteriä toimintamalleja, joissa jatkuva muutos on normi. Testaus on keskeinen osa sovelluskehitystä, mutta sen on mukauduttava jatkuvaan kehitykseen ja siksi sen toteuttamiseen tarvitaan automatisoituja ratkaisuja. Automaation avulla voidaan nopeuttaa prosesseja ja vähentää manuaalista työtä, laajentaa suoritettavien tehtävien määrää ja kehittää testausprosesseja ja -työkaluja vastaamaan kehityksen vaatimuksia.

Tässä diplomityössä esitellään testaustoimintamalleja ja -työkaluja ketterään web-sovelluskehitykseen pankkitoimialalla. Tutkimuksessa tarkastellaan pankkijärjestelmään kehitetyn sovelluksen kehitysprosessia ja etsitään ratkaisuja kehitystiimin kohtaamiin testaushaasteisiin.

Tutkielman aikana kokeiltiin käyttöliittymätestien ajamista ennen koodikatselmointia, mikä lisäsi testien ajoa ja kehittäjien kiinnostusta testien ylläpitoon. Tutkielmasta löydettyjä ratkaisuja voidaan soveltaa yrityksissä ja kehitystiimeissä kehitysmallin parantamiseksi ja sulauttamiseksi testaustoimintamalleihin.

Testaustoimintamallien sulauttaminen kehitysprosessiin voi parantaa tiimin työskentelymallia ja ehkäistä tehtävien siiloutumista kehittäjien välillä. Oikean järjestyksen varmistaminen kehitysprosessissa voi edesauttaa kehitystiimin tuottavuutta ja vähentää myöhään havaittavien ongelmien riskiä. Tämä tutkielma tarjoaa suuntaviivoja web-sovelluskehityksen kehitysmallille.

Avainsanat: automaatiotestaus, testaustyökalut, testausprosessit, web-sovellus, kehitysputki, Robot Framework, Jenkins

Keränen A. (2023) Automated Testing in Web Application Development. University of Oulu, Degree Programme in Computer Science and Engineering. Master's Thesis, 50 p.

ABSTRACT

Digitization is driving the development methods of financial industry applications towards agile operating models, where constant change is the norm. Testing is a key part of application development, but it needs to adapt to continuous development, which is why automated solutions are needed. Automation can speed up processes, reduce manual work, expand the number of tasks performed, and develop testing processes and tools to meet development requirements.

This thesis presents testing methods and tools for agile web application development in the banking industry. The study examines the development process of an application developed for a banking system and seeks solutions to testing challenges faced by the development team.

During the study, the running of GUI tests before code review was tested, which increased test runs and developers' interest in maintaining tests. The solutions found in the thesis can be applied in companies and development teams to improve the development model and integrate it with testing models.

Integrating testing models into the development process can improve the team's working model and prevent silos between developers. Ensuring the correct order in the development process can help increase the productivity of the development team and reduce the risk of late-detected issues. This thesis provides guidelines for the development model of web application development.

Key words: testautomation, testautomation tools, testprocess, web-based-sofware, pipeline, Robot Framework, Jenkins

SISÄLLYSLUETTELO

ALKULAUSE

Tämä diplomityö tehtiin yhdessä OP-Palvelut Oy:n kehitystiimin kanssa. Haluan kiittää OP-palvelut Oy:ta sekä kehitystiimini kollegoita tutkielmani aiheesta ja mahdollisuudesta kehittää tiimimme toimintamalleja. Erityiskiitokset FM Marko Siiralle ja DI Keijo Goman:lle asiantuntevasta tuesta tutkielman aikana nousseiden asioiden läpikäymisestä sekä niihin tarjoamasta avusta.

Haluan kiittää myös tutkimukseni ohjaajiani, TkT Lauri Lovénia sekä Dos. Susanna Pirttikangasta, tutkimuksen aiheen ja toteuttamisen tukemisesta.

Tämän diplomityön kielenhuollossa on käytetty ChatGPT-tekoälytyökalua.

Oulu, 4. 2023

Antti Keränen

LYHENTEIDEN SELITYKSET

GUI	Graphical User Interface, käyttöliittymä
DevOps	Development and Operations, kehitystyö ja operatiivinen toiminta
CI	Continuous Integration, jatkuva integrointi
CD	Continuous Delivery/Deployment, jatkuva julkaisu/jatkuva toimitus
PR	Pull Request, muutospyyntö

1. JOHDANTO

Digitalisoituminen on muuttanut finanssialaa ja tuonut mukanaan uusia työkaluja, jotka mahdollistavat nopeamman ja helpomman digitalisaation eri osa-alueilla [1]. Tämä kehitys on asettanut johtajille entistä suuremman roolin muutoksen hallinnassa. Vaikka digitalisaation ensimmäinen aalto on jo koettu, organisaatiot pyrkivät jatkuvasti kehittymään kohti entistä itseohjautuvampaa toimintaa [2].

Liiketoimintaa tukeva DevOps-toimintamalli mullistaa ohjelmistokehitystä kohti itseohjautuvuutta ja pyrkii poistamaan toiminnan siiloutumista. Se on synnyttänyt kulttuurisen muutoksen, jossa eri sidosryhmät työskentelevät joustavammin yhdessä, omaksuen automaation ja nopeutuvan tuotantosyklin [3]. Ketterät toimintamallit ja automatisaatio ovat levinneet muihinkin kuin ohjelmistotuotannon prosesseihin. Tämä tuotantoprosessien suoraviivaistuminen, halpeneminen ja oikea-aikaisuus on tuonut merkittäviä etuja liiketoiminnalle. Tavoitteena on tehdä asioita, joilla on iso vaikutus. [4]

Finanssialan muuttuessa yhä enemmän ohjelmistopainotteiseksi on tärkeää siirtyä ohjelmistotuotannosta tuttuihin prosesseihin, kuten jatkuvaan toimittamiseen, mikä vaatii jatkuvaa testausta. Laadunvarmistus on olennaisessa roolissa jatkuvan tuotannon eri vaiheissa, sillä ongelmien korjaaminen on kallista. Organisaatiot panostavat muun muassa testiautomaatioon kehitysprosessin laadun parantamiseksi. Parhaiten testaamisesta saadaan hyötyä, kun se suoritetaan oikein tehtyjen ja luotettavien testien avulla jatkuvan integraation ja jatkuvan toimittamisen julkaisun vaiheissa (CI/CD), jotka informoivat virheestä oikealla tavalla. Toimimattomat automaatiotestit vähentävät tulosten luotettavuutta ja aiheuttavat ylimääräisiä kustannuksia ja viiveitä. [5]

Verkkopankkien ohjelmistojen testaus on tärkeä osa kehitysprosessia, joka varmistaa ohjelmiston laadukkaan toteutuksen. Testaus suoritetaan useassa vaiheessa, alkaen sovelluskehityksen alkuvaiheista ja jatkuen pitkin sovelluksen elinkaarta. Koska verkkopankeissa käsitellään arkaluontoista asiakastietoa ja järjestelmät ovat alttiita hyökkäyksille, on toiminnallisuuksien valvonta erityisen tärkeää. Pankkitoimintaa myös säännellään tiukasti lainsäädännöllä, jota on noudatettava digitaalisia palveluita kehitettäessä. [6]

Jatkuvan toimittamisen keskeinen elementti on automaatio, joka vähentää manuaalista työtä ja nopeuttaa prosesseja. Kehittävien tiimien on luotava itselleen sopiva malli, jolla voidaan maksimoida tehokkuus ja samalla täyttää vaaditut kriteerit. Erilaisia työkaluja ja menetelmiä voidaan käyttää jatkuvan toimittamisen toteuttamiseen, ja jokaisen tiimin on hyvä suunnitella itselleen sopivat ratkaisut kehityksen eri vaiheiden toteuttamiseen. Pankkisovellusten kehittämisessä on erityisen tärkeää varmistaa, että automatisoidut testit kattavat sovelluksen kaikki toiminnallisuudet ja ovat luotettavia, jotta ohjelmiston laatu ja turvallisuus voidaan taata. [6]

1.1. Tutkimusongelma

Ohjelmistokehityksessä pyritään nykyään nopeuteen ja jatkuvuuteen, mikä tarkoittaa kehitystyövaiheiden automatisointia ihmisen manuaalisen työn vähentämiseksi. Silti lopputuotteen tai palvelun tulisi olla korkealaatuinen ja virheetön. Kehitystiimien tulee pystyä reagoimaan jatkuvasti muuttuviin asiakastarpeisiin, ja testaukseen liittyvät

tehtävät on pyrittävä suorittamaan yhä lyhyemmässä ajassa. Tämän vuoksi testaaminen tulisi hoitaa yhä enemmän automaation avulla. Erityisesti finanssialalla sovellusten virheettömyys on erittäin tärkeää, jotta asiakkaiden luottamus järjestelmiin säilyisi korkeana. Testaaminen on vain yksi osa kokonaisvaltaista laadunvarmistusta, joka kuuluu kaikkiin tuotannon vaiheisiin ja jonka vastuu kuuluu jokaiselle roolille. Jotta jatkuvuus säilyisi ohjelmistojen muuttuessa, laadunvarmistusta tulisi toteuttaa niin asiakasta kuin tuotekehittäjääkin ajatellen.

Ohjelmistokehityksen jatkuvan toimittamisen menetelmät ovat yhä suositumpia, ja niissä pyritään automatisoimaan mahdollisimman monia sovelluskehityksen vaiheita ja lisäämään laadullisia tarkistimia sovelluksen elinkaaren eri vaiheisiin. Tämä johtuu siitä, että tarkat laatukriteerit edellyttävät ohjelmistojen laadunvarmistuksen ennen niiden käyttöönottoa. Testausprosessi on tärkeä osa tätä laadunvarmistusta, ja siihen tarvitaan monipuolisia testausmenetelmiä, sillä virheitä ja ongelmia voi ilmetä monilla eri tavoilla. Erilaisten palveluntarjoajien tarjoamat testaus- ja analysointityökalut ovat tärkeä apu tässä prosessissa, ja niitä käytetään yhdessä tuottamaan arvokasta analyysiä tuotettavan ohjelmiston laadusta.

Ohjelmistokehityksessä testaaminen on keskeinen osa laadunvarmistusta ja tapahtuu usein koko ohjelmiston elinkaaren ajan. Testausta voidaan suorittaa eri tasoilla kooditasolta käyttöliittymän toiminnallisuuksien varmistamiseen. Testausmenetelmät jakautuvat analysointiin, manuaaliseen testaamiseen ja ajettaviin automaatiotestauksiin. Testaajalla on vastuu testauksesta, mutta manuaalitestauksen lisäksi ajanpuute voi estää riittävän testiautomaation ylläpidon ja kehittämisen. Tästä syystä automaatiotestauksen hyödyntäminen on tärkeää, sillä se voi vähentää testaamiseen käytettävää aikaa ja parantaa testien toistettavuutta.

Tämän diplomityön keskeinen tavoite on kuvata ja analysoida potentiaalisia automaatiotestausmenetelmiä finanssialan web-käyttöliittymäsovelluksen toimituskehykseen. Lisäksi tutkielma pyrkii ratkaisemaan ongelman, jossa testiautomaation ylläpidon ja kehittämisen vastuu jää yksittäisen testaajan vastuulle. Työssä käsitellään yleisiä web-sovelluksen testauksen vaiheita sekä käytännön automaatiotyökaluja, joiden avulla jatkuvan toimituksen kehitysmenetelmiä voidaan toteuttaa tehokkaasti ja luotettavasti.

1.2. Tutkimuksen tavoitteet

Tämän tutkimuksen tavoitteena on esitellä ohjelmistokehitysputken malli, jonka avulla voidaan toteuttaa web-pohjaisten pankkisovellusten laadukasta ja tehokasta automaatiota ohjelmistokehityksessä. Tätä mallia käsitellään tässä työssä myös nimellä kehitysputki. Lisäksi pyritään löytämään ratkaisumalleja, joilla jakaa testiautomaation ylläpito- ja kehittämisvastuuta useamman työntekijän vastuulle, välttäen yksittäisen testaajan työtehtävien siiloutumista.

Tämän tutkimuksen toisena tavoitteena on löytää ratkaisumalleja, jotka auttavat kehitystiimiä jatkuvassa toimittamisessa. Tämä vaatii stabiilien ja toimivien kehitysmenetelmien käyttöönottoa, joka voidaan integroida tiimin toimintamalliin. Ratkaisujen tavoitteena on nopeuttaa muutosten toimittamista samalla, kun ne täyttävät pankkialan vaatimukset. Tavoitteena on myös luoda ratkaisumalli, joka voidaan hyödyntää myös tiimin muiden ohjelmistojen kehityksessä. Tutkielman tuloksena löydettyjen ratkaisujen odotetaan vähentävän pitkän aikavälin "hukkaa" (engl. "waste") ja lisäävän kehitystiimin tuottavuutta.

Tutkielmassa tarkastellaan automaatiotestaamista web-sovelluksen kehityksessä finanssialan verkkopankki-järjestelmän pohjalta. Yhtenä lähtökohtana käytetään yhtä pankin ohjelmistoa, jonka avulla tutkimus tuo esiin yleisesti soveltuvia ratkaisuja web-sovelluksen testaukseen. Vaikka tutkimus käsittelee finanssialan verkkopankki-järjestelmää, käytetyt menetelmät ja työkalut ovat hyvin standardoituja ja laajasti käytettyjä myös muilla toimialoilla.

1.3. Tutkimuksen toteutus ja rakenne

Tämä diplomityö on toteutettu parantamaan kehitystiimin työskentelyä ja kehitysmenetelmiä. Yksityiskohtaisia sovellukseen tai yritykseen liittyviä asioita ei kuitenkaan käsitellä, sillä ne ovat luottamuksellisia tietoja. Tutkielmassa käsitellään yleisiä verkkosovellusten testaamiseen liittyviä työkaluja ja menetelmiä. Web-ohjelmiston testaaminen vaatii sekä automaation että manuaalisen työn yhteistyötä. Tutkimuksessa esitellään manuaalisen ja tutkivan testauksen menetelmiä, mutta painotus on enemmän automaation ja ohjelmistokehitysputken aikana käytettävissä järjestelmissä ja toiminnoissa. Tutkimus käsittelee myös muita web-sovelluskehityksen prosesseja ja työkaluja, mutta projektin testaamisprosessit rajataan kehitystiimin ohjelmistokehitysputkessa tapahtuviin prosesseihin. Tutkimuksessa ei käsitellä eri toimialojen tai pankkien sovellustestaamista, eikä tutkimuksessa tarkastella projektin luonteeseen liittyviä haasteita tai ongelmia.

Toisessa luvussa käsitellään ohjelmistokehityksessä laajasti käytössä olevat ohjelmistokehitysmenetelmät sekä yleisen ohjelmistokehitysputken malli. Kolmannessa luvussa käydään läpi yleiset testausmenetelmät web-sovelluksen elinkaaren aikana.

Neljännessä luvussa keskitytään verkkosovellusten testaamisessa käytettäviin työkaluihin ja niiden käyttötarkoituksiin, sivuten edellisessä luvussa mainittuja testausprosesseja.

Viidennessä luvussa esitellään web-sovellusta kehittävän tiimin käytössä oleva ohjelmistokehitysputki. Kuudennessa luvussa käydään läpi tutkielman havainnot ja pohdinnat. Seitsemäs luku on tutkielman yhteenveto.

2. OHJELMISTOKEHITYSPUTKI

Ohjelmistotuottajien on nykypäivänä pakko muuttaa toimintatapojaan joustavimmiksi ja nopeammiksi vastatakseen jatkuvasti muuttuviin kehitystarpeisiin. Tämän seurauksena ohjelmistotuotanto on kehittynyt automaatiomenetelmistä koostuvaksi vaiheittaiseksi malliksi, joka tunnetaan nimellä ohjelmistokehitysputki (engl. pipeline). Ohjelmistokehitysputken eri vaiheet suoritetaan automaattisesti koodimuutosten syntymisen yhteydessä. Tämä mahdollistaa koodimuutosten luotettavan toimittamisen tasaisin väliajoin. [7][8]

Ohjelmistokehitysputken automatisoinnin tavoitteena on vähentää riskejä, välttää inhimillisiä virheitä ja lisätä tiimin tuottavuutta [8]. Vaikka ohjelmiston toimittamisessa pyritään nopeuteen, testaaminen on silti suoritettava korkealla laadulla. Ketterät käytännöt pienentävät riskejä samalla, kun mahdollistavat tehokkaan ja nopean ohjelmiston toimittamisen. Automaation avulla manuaaliseen työhön käytetty aika vähenee, ja vaaditut tehtävät voidaan suorittaa automaattisesti. Automaatio mahdollistaa myös testaamisen monilla eri menetelmillä, jolloin ohjelmiston laatu pysyy korkealla nopeasti etenevistä muutoksista huolimatta. [7]

Ohjelmistokehitysputki koostuu yleensä neljästä vaiheesta: koodimuutoksesta, koodin kääntämisestä, koodin testaamisesta ja ohjelmiston asentamisesta tai toimittamisesta. Jokainen vaihe on riippuvainen edellisen vaiheen onnistuneesta suorittamisesta, ennen kuin seuraavaa vaihetta voidaan aloittaa. Sovellusta kehittävä tiimi seuraa ja raportoi vaiheiden onnistumista ja tuloksia. [8]

Ohjelmistokehitysputkien muoto vaihtelee ohjelmistojen ja ohjelmistotuottajien vaatimusten mukaan. Päävaiheet voivat sisältää erilaisia askelia, jotka liittyvät ohjelmiston toimittamiseen, testaamiseen tai asentamiseen. Testausvaiheita voi olla useampia, ja testaus ja kääntäminen voivat sisältää useita suoritettavia toimenpiteitä. Toimitus voi olla asennus tai muutoksen vieminen seuraavaan kehitysputkeen.

Esimerkiksi käyttöliittymäpohjaisten ohjelmistojen testaaminen edellyttää sovelluksen asentamista tarkoitukseen varattuun testiympäristöön. Käyttöliittymätestit on usein suoritettava ennen tuotteen toimittamista loppukäyttäjille. Vasta hyväksyttyjen käyttöliittymätestausten jälkeen voidaan suorittaa varsinaiset tuotantoon asennukset.

2.1. Kehitysputken menetelmät

Kehitysputken (CI/CD) menetelmien tarkoituksena on automatisoida ohjelmistotuotannon koko elinkaari, jotta mahdollisimman monet vaiheet saadaan toteutettua nopeasti ja luotettavasti [8]. Menetelmiin kuuluu useita vaiheittain käyttöönotettavia toimintoja, joiden avulla ohjelmistotuotanto voidaan automatisoida parhaimmillaan koodin luomisesta aina loppukäyttäjälle toimittamiseen saakka. [7]

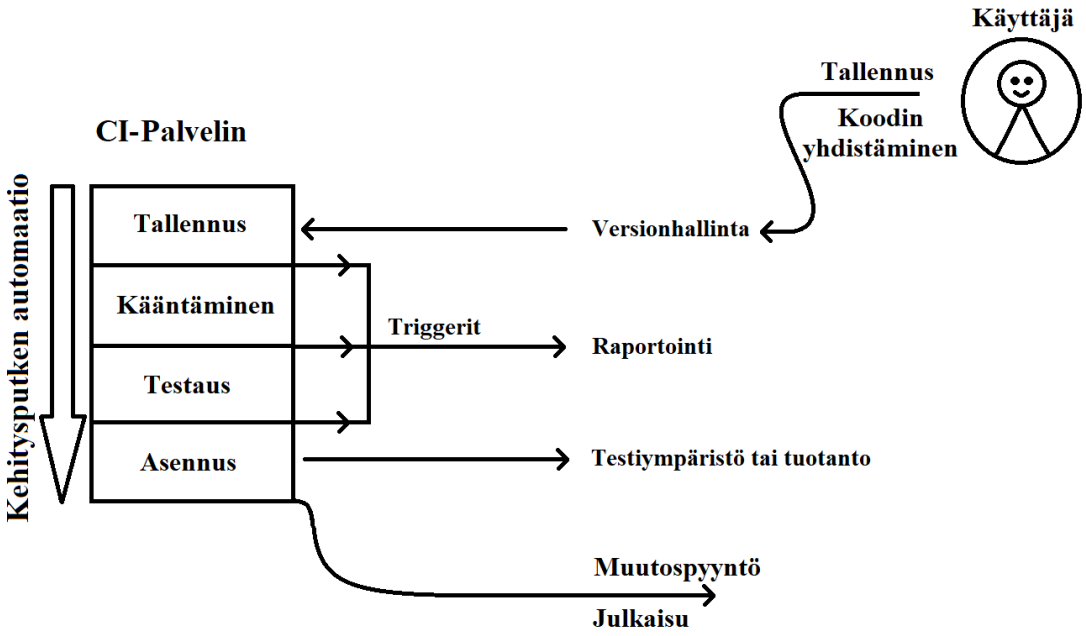
Jatkuva integraatio (engl. continuous integration, CI) on ohjelmistotuotannon menetelmä, jossa useamman kehittäjän koodimuutokset yhdistetään (”integroidaan”) automaattisesti aina muutosten jälkeen [9]. Tämä menetelmä mahdollistaa sen, että muutokset etenevät vaiheittain ilman taukoja, ja että toimivat ja toimimattomat muutokset siirtyvät edestakaisin pieninä palasina, rikkomatta kehityksen jatkuvuutta. Tutkimusten mukaan kehitysputken menetelmät ovat alalla paras ketjutustapa ja ne ovat laajalti käytössä. Menetelmät auttavat vähentämään kokonaisaikaa, joka kuluu

tuotteen toimittamiseen, vaikka niiden käyttöönottamisessa voi kohdata haasteita systeemin toimivuuden ja hallinnan suhteen. [10]

Jatkuva toimitus (engl. continuous delivery, CD) on menetelmä, jossa tiimit käyttävät automaatiotyökaluja suunnitellessaan, rakentaessaan, testatessaan ja julkaistessaan ohjelmistoa [11]. Tämä eroaa jatkuvasta integraatiosta, sillä jatkuva toimitus automatisoi sovelluksen asentamisen testiympäristöihin [10]. Jatkuva toimitus mahdollistaa nopean julkaisutahdin, alentaa kustannuksia ja aikaa sekä vähentää riskiä muutosten vaikutuksista, kun julkaisua lähdetään toimittamaan tuotantoympäristöön [11]. Toimitusmenetelmät ovat automatisoituja, jotta jokainen vaihe olisi nopea ja luotettava. Jatkuva toimitus on kehitysstrategia, joka automatisoi koodimuutosprosessin niin, että muutoksen vieminen versionhallintaan testataan ja varmennetaan automaattisesti. [12]

Jatkuva julkaisu (engl. continuous deployment, CD) on menetelmä, joka vaatii, että jokainen koodimuutos julkaistaan automaation avulla tuotantoon asti. Jatkuva julkaiseminen edellyttää myös, että ohjelmistokehitysputken muut osat ovat käytössä ja toimivat luotettavasti, jotta tuotantovalmius ja jatkuva käyttöönotto ovat mahdollisia. Tiimin pyrkimyksenä on pitää julkaisujen koko mahdollisimman pienenä ja pitää julkaisukynnys matalana. Jatkuva julkaiseminen tasaisin väliajoin tuo jatkuvasti arvoa tuotteen käyttäjille ja edistää tuottavan tahon jatkuvaa kehitystä. [7]

2.2. Kehitysputki palvelin



Kuva 1. Kehitysputki palvelimen toiminta

Kehitysputken automaatio pyrkii automatisoimaan koodimuutosten siirtämisen versionhallinnasta loppukäyttäjille. Koodimuutokset käyvät läpi monimutkaisen prosessin, ennen kuin ne ovat valmiita julkaistavaksi. Kuvassa 1 esitetään kehitysputki palvelimen (engl. CI-pipeline) toimintaa, joka suorittaa kehitysputken eri vaiheet. Vaiheisiin voi kuulua koodin kääntäminen (engl. build), staattinen koodianalyysi,

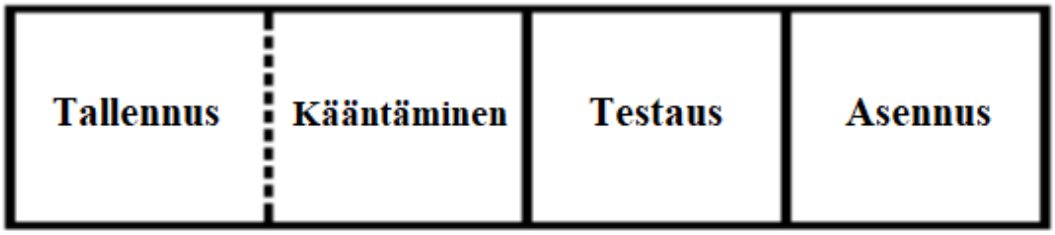
yksikkötestaus (engl. test), funktionaalinen testaus ja toimitukseen liittyvät toimenpiteet, kuten asentaminen (engl. deploy). Ennen prosessien suorittamista palvelimen tulee ensin kopioida (engl. clone) projektin koodi itselleen, jotta se voi suorittaa määritellyt toimenpiteet. [13]

Kehitysputki palvelimet käyttävät yleensä kolmannen osapuolen tarjoamia työkaluja, kuten versionhallinta-, koodinkääntö-, automaatio-, testiautomaatio- ja monitorointityökaluja. Ne tarjoavat keskitetyn kontrollin näiden työkalujen automaattiseen suorittamiseen. Kun eri työkalut yhdistetään kehitysputkeksi, koodin toimitusketju voidaan automatisoida valmiiksi verifioituna. [7]

Kehitysputki palvelin voidaan integroida versionhallintaan, jolloin se tunnistaa sisääntulevat muutospyynnöt ja aloittaa automaattisesti vaadittujen testien, analyysien ja käyttöönoton suorittamisen. Palvelimen tehtävänä on ilmoittaa, mikäli muutoksen läpikäyneet testit ovat hyväksytty ja muutos voidaan yhdistää projektin päähaaraan tai ottaa käyttöön. Muutosputki keskeytyy automaattisesti, mikäli ilmenee ongelmia, ja palvelin ilmoittaa niistä kehittävälle tiimille, joka korjaa ongelmat mahdollistaakseen muutosten läpiviennin. [14]

Tämä tutkielma esittelee yhden laajalti käytetyn palvelimen ohjelmistokehityksessä. Jenkins on palvelin, joka tarjoaa laajennettavissa olevan työkalupakin ohjelmiston toimitusketjun luomiseksi [13]. Muita vastaavia automaatiopalvelimia ovat muun muassa Travis CI ja Circle CI [14], mutta niitä ei tässä tutkielmassa eritellä tarkemmin, sillä ne palvelevat samaa käyttötarkoitusta.

2.3. Ohjelmistokehitysputkimalli



Kuva 2. Ohjelmistokehitysputken päävaiheet. (Kuva alun perin Shanika [8])

Kun kehittäjä tekee koodimuutoksen, hän tallentaa sen versionhallintaan. Versionhallinnan tietokanta ("repositorio", engl. repository) havaitsee uuden muutoksen ja käynnistää automaatiopalvelimen suorittamaan ohjelmistokehitysputkessa määritellyt toiminnot. Muutoksen voi myös laukaista koodihaarassa havaittu yhdistyminen, jossa kahden koodihaaran muutokset yhdistetään yhdeksi päähaaraksi. Automaatiopalvelin kuljettaa koodimuutoksen vaiheittain kehitysputken läpi, jossa ensimmäisenä suoritetaan yleensä koodin kääntäminen. Koodin kääntämisprosessissa käydään läpi koodin syntaksi ja muodostetaan koodista suoritettava binääri [15]. Yksikkötestausvaihe suoritetaan koodin kääntämisen jälkeen, jossa suoritetaan koodin logiikkaa käsittelevät yksikkötestit, jotka kehittäjä on itse toteuttanut sovelluksen toimintojen perusteella. Tähän asti koodimuutoksia ei ole tarvinnut asentaa mihinkään, ja koodin testaamisessa pysytään täysin kooditasolla. [8]

Kehittäjätiimillä on täysi vapaus määritellä kehitysputkssa tehtävät toimenpiteet. Koodille voidaan suorittaa useita käännöksiä, eri testaustasoja, useita asennuksia tai

vientejä eri kehitysputkeen tai ympäristöön. Todellisuudessa kehitysputki voi olla hyvin monimutkainen ja monitasoinen, jossa täytyy huomioida tarkat vaatimukset, kuten turvallisuus- ja suorituskykyvaatimukset. Kuvassa 2 on esitetty kehitysputken päävaiheet, jotka ovat yksinkertaistettu versio monimutkaisemmasta todellisuudesta. [8]

2.4. Jenkins-kehitysputki

Jenkins on alansa johtava avoimen lähdekoodin automaatiopalvelin, joka tarjoaa satoja erilaisia lisäosia koodin kääntämiseen, asentamiseen ja automatisointiin. Jenkins pystyy hallinnoimaan koko ohjelmistokehitysputkea, ja sen käyttöä on mahdollista laajentaa valmiilla lisäosilla (”plugineilla”) tai organisaatioiden itse kehittämillä ratkaisuilla. Jenkinsillä on jatkuvasti kasvava yhteisö, johon kuuluu kehittäjiä, testaajia, suunnittelijoita ja muita jatkuvasta integraatiosta, jatkuvasta toimituksesta sekä muista nykyaikaisista ohjelmistotoimitusmenetelmistä kiinnostuneita ihmisiä. [13]

Jenkinsin selainpohjainen käyttöliittymä mahdollistaa kaikkien kehitysputken osien hallinnan, järjestelyn ja seurannan, niiden konfiguroinnin ja ajamisen. Jenkinsin kautta voidaan esittää raportointia monella eri työkalulla suoritetuista tehtävistä ja testeistä. Ajettavien koodikäännösten ja tehtävien (engl. job) tapahtumalokien avulla käyttäjä voi ratkoa mahdollisia ongelmatilanteita. [13]

2.4.1. Jenkinsfile

Jenkins-kehitysputken määrittely tapahtuu yleensä projektin tekstitiedostossa nimeltä Jenkinsfile. Tämä tiedosto sijoitetaan projektin tiedostojen joukkoon helposti luettavaan paikkaan, josta versionhallinta pystyy käynnistämään CI-palvelimen suorittamaan kyseisen tiedoston sisältämän lyhyen ohjelman (”skriptin”). Versionhallinta käynnistää tiedoston laukaisemalla liipaisimen (”webhookin”), joka käynnistää koodin kääntämisen ja kertoo Jenkins-palvelimelle, minkä version ja koodihaaran (engl. branch) koodista tulee kääntää. Jenkins-palvelin hakee koodin versionhallinnasta ja lukee Jenkinsfile-tiedoston sisällön. Tämän jälkeen Jenkins suorittaa kehitysputken tehtävät, joiden tulokset ja tapahtumat näkyvät käyttäjälle. [13]

Jenkinsfile on kirjoitettu Apachen Groovy-koodikielellä [16]. Tiedoston alussa määritellään sen olevan kehitysputki ("pipeline"), jonka jälkeen määritellään ympäristömuuttujat. Nämä muuttujat sisältävät tietoja suorittavasta palvelimesta, kuten käytössä olevan sovelluksen kielen ja projektin tiedot, sekä webhook-osoitteen, joka on lisätty projektiin. Tiedostoon voidaan myös lisätä erilaisia hälytyksiä kehitysputken toiminnasta, joiden avulla kehittäjätiimi saa tietoa sen onnistumisista ja mahdollisista epäonnistumisista. [17]

2.4.2. Jenkinsfile – Vaiheet

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any

stages {
  stage('Build') {
    steps {
      sh 'echo "Ensimmäinen vaihe"'
    }
  }
  stage('Test') {
    steps
      sh 'echo "Testaus vaihe"'
    }
  }
  stage('Deploy') {
    steps {
      sh 'echo "Toimitus vaihe"'
    }
  }
}
```

Kuva 3. Esimerkki Jenkinsfilen vaiheet. (Esimerkki alun perin [16])

Jenkinsfilen vaiheet (engl. stage) koostuvat projektin kehittäjän itse määrittelemistä osa-alueista, kuten koodinkääntämisen, testaamisen ja asennuksen vaiheista. Kehittäjä voi muokata vaiheita kehitysputken käyttötarkoituksen mukaan, esimerkiksi dokumentaation luomiseen. Vaiheet asetetaan tiedostoon allekkain suoritusjärjestyksessä, ja komennot määritellään kunkin vaiheen sisällä. Vaiheet voidaan myös asettaa ehdollisiksi tai yhdenaikaisiksi, jolloin esimerkiksi asennuksen suorittaminen tapahtuu vain testiympäristöön ja käyttöliittymätestit jätetään asennuksen ulkopuolelle. Yhdenaikaisuuksia voidaan käyttää useamman testiajon asettamiseen ajettavaksi samanaikaisesti. Kuvassa 3 on esitetty esimerkki Jenkinsfilesta, joka sisältää useita eri vaiheita. [13][18]

3. WEB-SOVELLUKSEN TESTAUSPROSESSIT

Web-sovelluksen testaaminen on olennainen osa ohjelmistokehitysprosessia ja sillä on tärkeä rooli ohjelmiston laadun varmistamisessa. Tyypillisesti testaus suoritetaan eri vaiheissa, jotka ovat yksikkötestaus, integraatiotestaus ja hyväksyntätestaus. Tavoitteena on varmistaa tuotteen toiminnallisten ja ei-toiminnallisten ominaisuuksien toimivuus ja luotettavuus koodia muutettaessa. Testaukseen kuuluu myös testien suunnittelu, koodikatselmoinnit ja asianmukainen dokumentointi. Erityisesti finanssialan sovelluskehityksessä testaaminen on erityisen tärkeää, sillä pienikin virhe sovelluksessa voi johtaa vakaviin seurauksiin, kuten liiketoiminnan häiriintymiseen, tulojen menetyksiin tai mainehaittoihin. Vaikka testaus vie paljon aikaa kehitysprosessissa, sen poisjättäminen voi johtaa virheellisten sovellusten julkaisemiseen. [19][20]

Testausprosessit suositellaan aloitettavan ohjelmistokehityksen alkuvaiheessa, ja testauksen vaiheet muodostavat pyramidimallin testien kattavuudesta sovelluksen eri kerrosten välillä [19][21]. Pyramidin alimman kerroksen muodostavat yksikkötestit, jotka testaavat ohjelmiston logiikkaa kooditasolla, ja sovellus eritellään testattaviksi komponenteiksi. Välikerroksessa ovat integraatiotestit, jotka testaavat sovelluksen komponenttien ja sen käyttämien palveluiden väliset integraatiot. Sovelluksilla on yleensä myös käyttöliittymä, joka on yleensä web-käyttöliittymä. Pyramidin ylimmässä kerroksessa ovat käyttöliittymätestit, jotka testaavat sovelluksen toiminnallisuutta käyttöliittymältä käsin, varmistaen, että käyttöliittymän toiminnot toimivat käyttäjän näkökulmasta odotusten mukaisesti. Testausprosessin tavoitteena on varmistaa, että sovelluksen toiminnalliset ja ei-toiminnalliset ominaisuudet ovat verifioituja, jotta sovellus voidaan julkaista virheettömänä. [20]

Ohjelmiston testaaminen on monivaiheinen prosessi, joka sisältää sekä automaatiotestauksen että manuaalisen testauksen. On tärkeää tunnistaa, mitkä osat kannattaa automatisoida ja mitkä on hyvä testata manuaalisesti. Erityisesti web-pohjaisten sovellusten testaaminen eroaa perinteisten sovellusten testaamisesta, varsinkin jos sovellus tukee jatkuvasti muuttuvaa liiketoimintaa. Jatkuvasti muuttuville sovelluksille onkin tärkeää kehittää tehokkaat testausprosessit, jotka voidaan suorittaa nopeasti ja luotettavasti.

Tutkiva testaus on manuaalista testausta, joka ei perustu dokumentoituihin testitapauksiin vaan testaajan kokemukseen ja intuitioon. Testaaja suunnittelee, suorittaa ja raportoi testinsä samanaikaisesti. Tutkiva testaus soveltuu erityisesti tilanteisiin, joissa testattavan tuotteen ominaisuudet ovat vaikeasti dokumentoitavissa tai muuttuvat nopeasti, kuten web-sovellusten tapauksessa. Web-sovellusten testaaminen on haastavaa, sillä ne ovat usein hajautettuja järjestelmiä, joiden toimintaa on vaikea ennustaa etukäteen. [22][23][24]

Tässä tutkielmassa esitellään ne testausprosessit, jotka soveltuvat DevOps-käytäntöjen mukaisesti kehitettävän web-ohjelmiston testaamiseen. Siihen kuuluu testaus sovelluksen koko elinkaaren ajalle kehitysvaiheista eteenpäin. Jatkuva testaaminen ja kehittäminen luovat yhdessä jatkuvan toimittamisen kyvykkyyden.

3.1. Kehitysvaiheen testaaminen

Ohjelmiston kehitysvaiheinen testaus on monivaiheinen prosessi, jossa useat henkilöt ja työkalut varmistavat ohjelmiston laadun. Testausprosessi sisältää useita pieniä

vaiheita, jotka voivat vaihdella yrityksittäin riippuen esimerkiksi siitä, kuinka monta henkilöä on mukana kehitysprosessissa tai mitkä ovat yrityksen standardit ja vaatimustaso.

3.1.1. Katselmointi (Review)

Katselmointi on ensimmäinen askel ohjelmiston laadunvarmistuksessa, jossa kehittäjä antaa koodimuutoksensa toisen henkilön katselmoitavaksi. Tämä tapahtuu yleensä projektin käyttämän versionhallintatyökalun avulla. Esimerkiksi Bitbucket-työkalu tarjoaa käyttöliittymän koodikatselmointiin ja versionhallintaan [25]. Jos toinen henkilö havaitsee koodia katsellessaan puutteita tai asioita, joita voitaisiin tehdä paremmin tai toisella tavalla, hän voi palauttaa ohjelmiston kehittäjälle. Lopuksi kehittäjä ja katselmoija voivat yhdessä sopia ratkaisuista, jotka voidaan ottaa mukaan kehitteillä olevaan versioon. Näin ollen kehittäjä ei voi viedä koodimuutoksia eteenpäin kehitysputkessa ilman, että vähintään yksi toinen henkilö on hyväksynyt sisään tulevat koodimuutokset.

Katselmointi mahdollistaa tiedon ja vastuun jakamisen useammalle henkilölle, mikä estää koodituntemuksen keskittymisen vain yhdelle henkilölle. Katselmoitavat muutokset pyritään pitämään pieninä, jotta koodin ymmärtäminen ja katselmointi on helpompaa [26].

3.1.2. Yksikkötestaus

Yksikkötestaus on olennainen osa nykyaikaista ohjelmistokehitystä. Yksikkötestit auttavat kehitysvaiheessa tarkistamaan yksityiskohtaisesti koodin ominaisuuksien toimivuuden eristämällä sen täysin muista. Tämä mahdollistaa ohjelmoijien testata omia koodimuutoksiaan, mikä lisää kehitystyön tehokkuutta ja auttaa varmistamaan ohjelmiston laadun jo varhaisessa vaiheessa. [27][28]

Yksikkötestaus soveltuu erinomaisesti jatkuvan ohjelmistokehityksen alkuvaiheisiin. Ohjelmoijat voivat itse luoda testitapauksia samalla, kun he kirjoittavat koodia ja ylläpitävät testejä. Yksikkötestien kattavuutta voidaan mitata, mikä helpottaa kehittäjiä havaitsemaan ne koodialueet, joilla on puutteita toimintojen verifioinnin osalta. Testitapaukset tarkistavat koodin toiminnallisuuden jo ennen varsinaista asennusta testausympäristöön, joten ensimmäiset koodivirheet voidaan havaita koodimuutoksen varhaisessa vaiheessa.

Tutkimuksessa yksikkötestaus nähdään yhtenä ohjelmistokehitysputken vaiheista. Testit suoritetaan koodin käännösprosessin jälkeen, ja jos testit eivät mene läpi, kehitysprosessi pysähtyy. Tässä tutkielmassa esitellään yksi laajalti käytetty työkalu, jota käytetään myös kyseisen tutkimuksen yhteydessä luodun testausmallin yhtenä testausvaiheena.

3.1.3. Koodianalysointi

Koodimuutosten analysointi eri työkalujen avulla ennen niiden yhdistämistä sovelluksen päähaaraan on tärkeä osa ohjelmistokehitystä. Analyysityökalut (”skannerit”, engl. scanner) ja yksikkötestit analysoivat tehdyn koodimuutoksen ja raportoivat koodarille suoraan havaituista puutteista. Tämän analyysin perusteella

uuden koodin yhdistämistä päähaaraan voidaan evätä, jos puutteellisuudet kasvavat liian suuriksi. Tällä tavoin vältetään mahdollisten virheiden aiheuttamat häiriöt sovelluksen toiminnassa ja varmistetaan ohjelmiston laatu ennen sen julkaisua.

Staattisten koodianalysointityökalujen tarkoituksena on havaita yleisesti tunnistettuja ongelmia lähdekoodista. Nämä ongelmat voivat liittyä koodaustapaan, tietoturvaan tai käytettyihin komponentteihin ja niiden versioihin. Koodianalyysi perustuu sääntöihin, joita työkalujen käyttäjät voivat myös itse määritellä. On tärkeää havaita epäkohdat jo ohjelmistokehityksen alkuvaiheessa, jotta ne eivät aiheuta lisäkustannuksia myöhemmissä vaiheissa.

Kun koodi on analysoitu ja hyväksytty etenemään varsinaiseen testaukseen, koodimuutokset tuodaan testattavaksi testausta tekevälle henkilölle. Testaaja käy läpi uuden toiminnallisuuden sekä vanhat toiminnallisuudet perustuen sovelluksen tuntemukseensa. Uusi toiminnallisuus testataan sitä varten laadittujen tavoitteiden mukaisesti, jotta voidaan varmistaa sen toimivuus ja yhteensopivuus sovelluksen kanssa.

3.1.4. Manuaalitestaus

Taulukko 1. Manuaalitestaus vs automaatiotestaus

Manuaalitestaus	**Automaatiotestaus**
Tutkiva testaus	Regressiotestaus
Testitapaukset voidaan suunnitella suorituksen aikana	Testitapaukset ovat aina määritelty ja kirjoitettu etukäteen
Suorittamiseen kuluu enemmän aikaa	Suorittamiseen kuluu vähemmän aikaa
Sama testitapaus ajetaan kerran	Samaa testitapausta ajetaan useasti
Testauksen luonne: Ad-hoc Testaustavat voivat vaihdella	Testauksen luonne: Stabiili Testaustavat pysyvät samana
Suorittamiseen kuluu enemmän aikaa	Suorittamiseen kuluu vähemmän aikaa
Testi voi sisältää ihmisestä johtuvia virheitä	Testi toteutetaan aina samalla tavalla
Suurempi todennäköisyys löytää uusia havaintoja	Mahdollisuus löytää vain tiedostettuja ongelmia
Lyhyen aikavälin kustannukset pienenevät	Pitemmän aikavälin kustannukset pienenevät

Manuaalinen testaus on tärkeä osa ohjelmistokehitystä, jossa sovelluksen toimivuutta testataan käsin. Testaus vaatii testaajan työpanosta ja vie aikaa, ja sen tarkoituksena on varmistaa ohjelmiston toimintojen oikeellisuus [19][23]. Manuaalitestaus toteutetaan yleensä sovelluksen käyttöliittymällä, joko selaimen avulla verkkosovelluksille tai sovellukselle kehitetyn käyttöliittymän avulla paikallisesti toimiville sovelluksille. Uusien toiminnallisuuksien sekä tutkivan testauksen suorittaminen on ketterämpää suorittaa manuaalisesti, sillä automaatio vaatii aluksi aina kehitystä. Manuaalisen testauksen voi myös suunnitella etukäteen testaussuunnitelmalla, mikä auttaa testauksen seuraamisessa ja tulosten dokumentoinnissa. Manuaalitestauksen huonona puolena on kuitenkin sen hitaus ja rajallinen kapasiteetti, mikäli tarvitaan toistoja ja testattava kokonaisuus on laaja. Testitulosten luotettavuus voi kuitenkin olla parempi kuin automaatiotesteillä tuotettujen tulosten. Mikäli automaatiotestien suoritus ei onnistu tai niitä ei ehditä korjata ajallaan, manuaalitestauksella voidaan tehdä korvaava työ. Kaikki ohjelmistot vaativat manuaalitestausta ennen kuin niille voidaan toteuttaa koneella suoritettavia toiminnallisia testejä. [19][22][23][29]

3.1.5. Tutkiva testaus

Tutkiva testaus on ihmisen tekemää manuaalista testausta, jonka tarkoituksena on löytää virheitä, joita ei ole huomioitu suunnitteluvaiheessa. Tällaisia virheitä voivat olla esimerkiksi käytettävyysongelmat, järjestelmän toimintaan liittymättömät virheet tai sellaiset virheet, jotka eivät ilmene perinteisessä testauksessa. Tutkiva testaus perustuu kokeiluun ja sen periaatteena on, että testit syntyvät samalla kun testausta suoritetaan. Tutkiva testaus ei vaadi laajaa suunnittelua tai dokumentointia, mutta sen tulokset on raportoitava ohjelmistokehitystiimille. Tutkiva testaus on erityisen tärkeää sovellusten käytettävyyden ja toiminnan kannalta. [29]

3.1.6. Regressiotestaus

Ohjelmiston elinkaaren aikana on tärkeää varmistaa, että vanhat toiminnallisuudet toimivat edelleen uusien koodimuutosten jälkeen [23]. Tämä prosessi tunnetaan regressiotestauksena, jonka tarkoituksena on varmistaa, että uudet muutokset tai ohjelmiston ympäristön muutokset eivät vaikuta ohjelmiston toiminnallisuuksiin tai sen määriteltyjen vaatimusten täyttämiseen [23].

Regressiotestausta voidaan ajatella vanhojen toiminnallisuuksien testaamisena, ja sitä voidaan tehdä sekä manuaalisesti että automaation avulla. Automaatio nopeuttaa regressiotestauksen prosessia aiemmin kehitettyjen testien avulla, kun taas manuaalitestauksella varmistetaan toiminnallisuudet, jotka saattavat jäädä automaatiotestauksen ulottumattomiin. On kuitenkin tärkeää huomata, että ohjelmiston kehityksen aikana myös vaatimukset voivat muuttua, ja siksi on tärkeää päivittää testitapaukset vastaamaan uusia vaatimuksia. [29]

3.1.7. Funktionaalinen testiautomaatio

Verkkosovellusten käyttöliittymien testaamisessa käytetään yhä enemmän työkaluja, jotka mahdollistavat toiminnallisuuksien testaamisen automaattisesti. Tämä johtuu siitä, että web-sovellukset ovat yhä monimutkaisempia ja julkaisuajat lyhyempiä, mikä pakottaa ohjelmistotuottajat hyödyntämään automaatiota laadunvarmistuksessaan. [30]

Automaattiset testit testaavat sovelluksen toiminnallisuuksia pääasiassa käyttöliittymän kautta, esimerkiksi käyttämällä selainta, ja ne suoritetaan testausympäristössä, johon sovellus asennetaan. Automaatiotestauksessa voidaan rakentaa samoja testitapauksia kuin ihmisen käyttöliittymällä tekemät toiminnot, mikä vähentää manuaalista työtä ja nopeuttaa kehitysprosessia, kun koodia muutetaan vanhojen toiminnallisuuksien ollessa jo käytössä. Automaatiotestausta pyritään kehittämään jatkuvasti sovelluksen saadessa uusia toiminnallisuuksia, ja testien kattavuuden ollessa korkea, niiden hyöty kasvaa. Kuitenkin laajan testikokonaisuuden ylläpitoon ja kehittämiseen kuluva aika voi koitua ongelmaksi. [24]

Automaatiotestit voidaan kytkeä myös kehitysputkeen, asennusvaiheen jälkeen, jolloin ne auttavat verifioimaan koodimuutoksista aiheutuvaa regressiota. Automaatiotestien kehittäminen ja ylläpito vie paljon aikaa, ja siksi on tärkeää valita oikea automaatiokattavuuden ja ajankäytön suhde. Automaatio voi kuitenkin vähentää manuaalisten toistojen määrää ja nopeuttaa pitkällä aikavälillä testausvaiheita. Automaation tarkoituksena on siis poistaa ylimääräinen ajan käyttö yksinkertaisesti

toistettavista tehtävistä, jotta ihmistestaajan aikaa säästyy luovaan testaamiseen, eikä korvata sitä. [24]

3.1.8. Hyväksymistestaus

Hyväksymistestaus suoritetaan yleensä silloin, kun ohjelmisto on saavuttanut julkaisukelpoisen tilan kehitysvaiheessaan [29]. Tällöin testauksessa ei enää tehdä improvisoituja testitapauksia, vaan testauksessa suoritetaan sille ennalta määritellyt testitapaukset. Hyväksymistestaus suoritetaan ympäristössä, joka pyritään mallintamaan ohjelmiston tulevan käyttöympäristön mukaiseksi, ja sen tarkoituksena on varmistaa ohjelmiston toimivuus mahdollisimman realistisessa käyttöympäristössä.

Vaikka jossain käyttötapauksissa loppukäyttäjä saattaa suorittaa hyväksymistestauksen, finanssialan sovellusten testausta ei voida jättää pelkästään loppukäyttäjälle, vaan kaikkien ohjelmistojen toimivuus on varmistettava ennen kuin sovellus käsittelee oikeaa dataa. Tästä syystä testaus tapahtuu ns. alpha-testauksena [29]. Hyväksymistestaukseen kuuluu manuaalitestauksen lisäksi myös muita testausalueita, kuten funktionaalinen automaatiotestaus, kuormitustestaus ja tietoturvatestaus.

3.2. Kuormitustestaus

Kuormitustestaus on tärkeä osa hyväksymistestausprosessia, jossa arvioidaan sovelluksen suorituskyky ja kapasiteetti sen asetettuihin käyttöastevaatimuksiin perustuen. Tarkoituksena on ymmärtää järjestelmän toimintakyky korkean kuormituksen alla ja kerätä tietoja, jotka auttavat parantamaan sen suorituskykyä. [31]

Kuormitustestauksen suunnittelussa on tärkeää tietää ohjelmiston tuleva käyttäjämäärä sen käyttövaiheessa. Testaus on suunniteltava sovelluskohtaisesti ja sen on vastattava sille asetettuja vaatimuksia, jotta mahdolliset pullonkaulat voidaan tunnistaa ja korjata. [31]

3.2.1. Kuormitustestauksen alalajit

Load-testaus antaa käsityksen verkkosovelluksen toimivuudesta sen käyttöolosuhteissa simuloiden käyttäjien aiheuttamaa kuormaa odotetulla käyttöasteella. Tällä testillä pyritään löytämään pullonkauloja ennen sovelluksen julkaisua. Testien kestoa ja käytettyä kuormaa voidaan säätää vastaamaan sovelluksen käyttötarkoitusta. Load-testaus on kaikista kuormitustestaustyypeistä tärkein.[32]

Stress-testauksessa sovellusta kuormitetaan enemmän kuin oletettu käyttöaste, jotta voidaan testata sovelluksen vakautta suuremman kuorman alla. Tämä auttaa havaitsemaan ongelmia, jos kuormitus kasvaa oletettua suuremmaksi.[32][33]

Piikkitestausta, jota kutsutaan myös nimillä peak- tai spike-testaus, käytetään silloin kun halutaan testata sovelluksen käyttäytymistä lyhyellä aikavälillä erittäin korkean kuorman alla. Tarkoituksena ei ole jatkaa ylimitoitettua kuormitusta, vaan testata, miten sovellus käyttäytyy poikkeuksellisessa käyttötilanteessa, esimerkiksi ruuhkan tai poikkeuksellisen käyttäjätulvan aikana. Piikkitestaus voi myös auttaa varmistamaan sovelluksen toiminnan DOS-tyyppisten tietoturvahyökkäysten varalta. [33]

Endurance-testauksella testataan sovelluksen suorituskykyä pitkän ajan kuluessa normaalin käyttöasteen alla. Tämän testin avulla voidaan havaita ongelmia, jotka voivat johtaa suorituskyvyn huononemiseen ajan myötä, kuten muistivuodot. [33]

Skaalautuvuustestauksessa testataan ohjelmiston suoriutumista erilaisissa olosuhteissa, joissa kuormaa voidaan kasvattaa tai resursseja vaihtaa. Tämän testin avulla voidaan arvioida ohjelmiston suorituskykyä muuttuvissa käyttötilanteissa.[33]

Poikkeustilannetestauksessa testataan ohjelmiston käyttäytymistä erilaisissa virhetilanteissa. Tämä testi arvioi ohjelmiston kykyä käsitellä virheitä ja varmistaa, että se toimii oikein poikkeuksellisissa tilanteissa. Tärkeitä asioita ovat esimerkiksi komponenttien välisten yhteyksien kesto, uudelleenyritykset, fallback-toiminnot ja palautuminen. [34]

3.3. Tietoturvatestaus

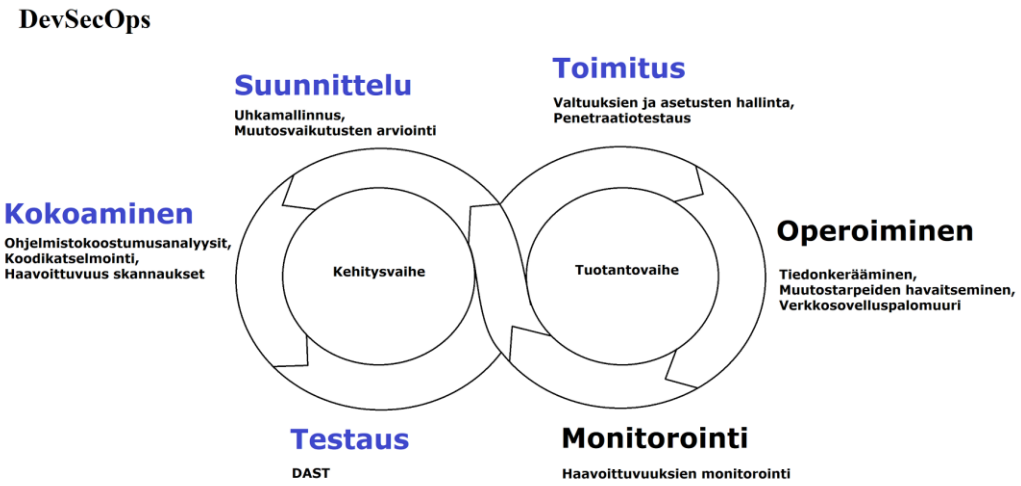
Verkkosovellusten tietoturva on tullut yhä tärkeämmäksi kehittyvän teknologian ja datan määrän kasvun myötä. Tietoturvaan liittyy monia tutkittavia osa-alueita, sillä mahdollisuus hyökkäyksiin ja tietojen väärinkäyttöön kasvaa jatkuvasti kehittyvien kriittisten palveluiden myötä. [35]

EU-maissa alettiin vuonna 2018 soveltaa yleistä tietosuoja-asetusta, joka säätelee henkilötietojen käsittelyä ja määrittelee henkilötietojen käsittelyyn liittyvät oikeudet ja vapaudet. Ohjelmistokehityksessä henkilötietojen käsittely tapahtuu sekä manuaalisten että automaattisten prosessien avulla, jolloin tietojen käsittelyssä on noudatettava asetuksessa määriteltyjä sääntöjä. Asetuksen tarkoituksena on suojella henkilötietoja ja niiden liikkuvuutta sekä turvata luonnollisten henkilöiden perusoikeudet ja vapaudet. [36]

Finanssialalla tietoturva on yksi tärkeimmistä vaatimuksista tietojärjestelmissä, ja sen huomioiminen on tärkeää läpi sovelluskehityksen elinkaaren. Koska yksikään järjestelmä ei ole täysin tietoturvallinen, pankkipalveluiden osalta täydellisyyttä tavoitellaan jatkuvasti. Suomessa pankit hallinnoivat itse omia turvallisuusratkaisujaan, ja ne ovatkin digitaalisen tunnistamisen edelläkävijöitä. Tietoturvaratkaisujen hajauttaminen useamman toimijan varaan voi myös tuoda lisäturvaa. [37]

Tietoturvallinen järjestelmä on tärkeä ansaitun asiakasluottamuksen kannalta, ja häiriö tietojärjestelmässä voi aiheuttaa pitkäaikaisia haittoja maineelle, taloudelliselle tuotolle ja asiakasluottamuksen palauttamiselle. [37]

3.3.1. DevSecOps



Kuva 4. DevSecOps. [38]

DevSecOps (ks. Kuva 4) on organisaatiollinen ja tekninen metodologia, joka hyödyntää automaattisia työkaluja projektin hallinnan prosesseissa. DevSecOps integroi aktiiviset tietoturva-auditoinnit ja testaukset ketterään kehittämiseen, jotta sovellus voidaan rakentaa alusta alkaen turvalliseksi, eikä turvallisuutta tarvitse lisätä vasta jälkikäteen.[38]

Tietoturva pyrkii seuraamaan modernin ohjelmistokehityksen jatkuvan toimittamisen menetelmiä. Kehitysprosessiin integroimalla tietoturvan toteuttaminen voidaan jalkauttaa ohjelmiston elinkaareen, ja siten tietoturva voidaan ajatella olevan aktiivisena osana automaattista kehitysputkea. Tavoitteena on minimoida tietoturvahaavoittuvuuksien riskit, varmistaa operaatioiden ja henkilöiden toimintatavat ja automatisoida mahdollisimman monet kehitysprosessin vaiheet. Näin sovellus voidaan rakentaa vastuullisesti ja parhaista käytännöistä noudattaen, samalla nopeuttaen ohjelmistokehityksen läpimenoaikaa. [38]

3.3.2. Static application security testing – SAST

Static Application Security Testing (SAST) -työkalut analysoivat lähdekoodia staattisesti etsien mahdollisia haavoittuvuuksia komponenteista. Tietoturvahaavoittuvuuksien havaitseminen kehitysvaiheen alkuvaiheessa on tärkeää, sillä korjaaminen myöhemmin elinkaaren aikana on vaikeampaa ja kalliimpaa. Staattinen koodin analysointi voidaan integroida myös kehitysprosessiin, jolloin analysointi tapahtuu automaattisesti järjestelmän havaitessa tulevia muutoksia projektiin. Analysointia voidaan tehdä myös erillisillä tietoturva-auditoineilla, joissa projektin ulkopuolinen taho käy läpi lähdekoodia työkalujen avulla vielä tarkemmin tietoturvan osalta. [39]

3.3.3. Software composition analysis - SCA

Ohjelmiston koostumusanalyysin tarkoituksena on löytää haavoittuvuuksia ja lisenssejä avoimen lähdekoodin komponenteista, joita käyttäessä voi olla riski kehitettävän ohjelmiston tietoturvallisuudelle [40]. Tämä analyysi tarkistaa projektin riippuvuudet mahdollisten tietoturvahaavoittuvuuksien varalta. SCA-työkalut tarkistavat projektissa käytetyt avoimen lähdekoodin komponentit ja auttavat valitsemaan versioita ja komponentteja, joissa havaitut haavoittuvuudet on jo korjattu, välttäen siten potentiaalisia tietoturvaongelmia. [41]

Integroimalla nämä käytännöt ja analyysit ohjelmiston kehitysvaiheeseen, pyritään varmistamaan mahdollisimman kattava analyysi käytettävistä menetelmistä ja haavoittuvuuksista. Tämä helpottaa nopeaa etenemistä kehityksessä samalla kun menettelyt ja käytännöt huomioidaan kattavasti ja luotettavasti. Testien integroiminen kehitysputkeen on siis oleellinen asia kehitysprosessissa, jotta tietoturva voidaan huomioida tehokkaasti ja mahdollisimman aikaisessa vaiheessa. [40]

4. WEB-SOVELLUKSEN AUTOMAATIOTESTAUSTYÖKALUT

Automaatiotestauskehys on ohjelmakehys, joka mahdollistaa automaatiotestauksen perustoiminnot, kuten selaimen avaamisen ja testien suorittamisen ja hallinnan. Testauskehyksiä on monenlaisia, kuten modulaarisen testauksen kehys, testikirjastokehys, avainsanalähtöinen testauskehys, dataohjattu testauskehys ja hybridi automaattinen testauskehys. [27][28]

Modulaarinen automaatiotestikehys luo testattavalle sovellukselle tyypillisesti pieniä, itsenäisiä ja uudelleenkäytettäviä moduuleja, fragmentteja ja komentosarjoja. Tämä kehys on helpoin käyttää ja ylläpitää samalla, kun se parantaa testiohjelmistojen ylläpidettävyyttä ja skaalautuvuutta. [28]

Testikirjastokehys on samanlainen kuin modulaarinen kehys, mutta hajottaa testattavan sovelluksen mieluummin proseduureiksi ja funktioiksi kuin skripteiksi. Testikirjastokehys edellyttää uudelleenkäytettävien moduulien, fragmenttien ja funktiokirjastotiedostojen luomista testattavana olevaa sovellusta varten. [28]

Avainsanoihin perustuva testaus on sovelluksesta riippumaton automaatiokehys, joka vaatii datan ja avainsanojen kehittämistä testattavaa sovellusta varten. Dataohjattu testikehys toimii samalla periaatteella kuin avainsanoihin perustuva testaus, mutta antaa työkalulle käytettävän datan tiedostona, josta se luo suoritettavan koodiskriptin. Hybridiautomaattinen testikehys yhdistää kaikki edellä mainitut kehystyypit, hyödyntäen niiden vahvuuksia ja korjaten niiden puutteita. [28]

4.1. Dynaamiset yksikkötestaustyökalut

Taulukko 2. Selainpohjaiset testiautomaatiokehykset web-sovelluksille

Työkalu	**Avoimen lähdekoodin työkalu**	**Ohjelmointikielet**	**Staattinen/dynaaminen**	**Soveltuu kehitysputki-testaukseen**
JUnit	Kyllä	Java	Dynaaminen	Kyllä
MochaJS	Kyllä	JavaScript / Node.js	Dynaaminen	Kyllä
Jasmine	Kyllä	JavaScript / Node.js	Dynaaminen	Kyllä
Jest	Kyllä	Babel, TypeScript, Node. js, React, Angular, Vue. js, Svelte	Dynaaminen	Kyllä

Yksikkötestauksen tarkoituksena on pitää koodi puhtaana ja yksinkertaisena, kun koodimuutosten määrä kasvaa. Yksikkötestausmenetelmät testaavat lähdekoodin osat yhdessä ja erikseen [42]. Yksikkötestaustyökalut ovat usein kolmannen osapuolen työkaluja, jotka helpottavat testausprosessia [42]. Testien tärkeys korostuu testaustavoissa ja testien laadussa. Huonot testit eivät edistä kehitysprosessia, mutta

hyvin laaditut testit parantavat koodin laatua. Hyvä testi havaitsee virheet muutosten tapahtuessa. Mitä enemmän koodia syntyy, sitä tärkeämpää testaus on monimutkaisuuden ja ymmärrettävyyden kannalta. [43]

Dynaamiset testauskehykset toimivat testaajan tekemien testien mukaisesti. Testit vaativat kehitysosaamista ja koodiosaamista, jotta oikeanlainen testaaminen on mahdollista. Tässä tutkielmassa käydään läpi yleisimmin web-sovelluskehityksessä käytettyjä testityökaluja.

4.1.1. JUnit

JUnit on yksinkertainen avoimen lähdekoodin testauskehys, jota käytetään yksikkötestaamiseen [44]. Se on laajasti käytetty testauskehys Java- sekä käyttöliittymäohjelmistokehityksessä, ja vuonna 2018 se oli suosituin Java-kielinen testauskehys [45]. Lähes 97 % java-projekteista käytti JUnitia jossain kohtaa projektin elinkaarta [45]. JUnitin suosittu käyttötarkoitus on luoda sarja yksikkötestejä, jotka voidaan suorittaa automaattisesti, kun ohjelmistoa muutetaan [46]. Kehittäjät voivat luoda testitapauksia, joilla voidaan verifioida koodilogiikasta odotettuja vasteita koodin palauttamiin oikeisiin vasteisiin. Testit voidaan kirjoittaa ennen varsinaisen ohjelman toteutusta, jolloin toteutuksen voi tehdä testejä vasten. JUnitin suosio kasvoi ja laajeni nopeasti sen ilmestymisen jälkeen. [46][47]

JUnit on kehitetty Javalle, ja testit suoritetaan kahdessa päävaiheessa: konfigurointivaiheessa ja testien suoritusvaiheessa. Konfigurointivaiheessa rakennetaan testin objectihierarkia, joka muodostuu TestCase nimisen alaluokan instansseista, joissa jokainen objecti kuvaa yhtä testitapausta. Testit suoritetaan testien suoritusvaiheessa siinä järjestyksessä, kun ne on luotu, ja tulokset nauhoitetaan TestResult objectiin. Analysointi ja tulosten raportointi -vaiheissa tulokset näytetään käyttäjälle visuaalisesti. [46][47]

JUnit erottaa virheet ja viat toisistaan. Testien raportoimat virheet indikoidaan AssertionFailureError instanssilla ja muut viat ilmenevät erroreina. Molemmat käsitellään Javan poikkeuksina ja kerätään TestFailure objectiin. [46][47]

4.1.2. JasmineJS

Jasmine on avoimen lähdekoodin testauskehys, joka noudattaa käyttäytymispohjaisen kehityksen prosessia tarkastaen jokaisen koodirivin olevan yksikkötestattu [47]. Jasmine ei ole riippuvainen muista JavaScript-testauskehyksistä ja se toimii ilman DOM:ia. Jasmine on saanut vaikutteita Rspec-, JS Spec- ja Jspec-testauskehyksistä, ja on saatavilla eri versioissa, kuten stand-alone, ruby gem ja Node.js -yhteensopivana. [47]

Jasminen pidetään helposti opittavana ja hyvin dokumentoituna sen pitkän historian ansiosta. Siitä löytyy runsaasti tietoa hakukoneilla. Jasmine on yhteensopiva lähes kaikkien kehysten ja kirjastojen kanssa, mikä tekee siitä joustavimman JavaScript-testauskehyksen. Jasminen syntaksi on puhdasta [48]. Lisäksi useat CI-serverit tukevat Jasmine-kehystä. On kuitenkin huomioitava, että Jasmine ei tue asynkroonista testausta ja sen konfigurointi voi olla haastavaa. [49]

4.1.3. MochaJS

Mocha on Node.js:n päälle rakennettu JavaScript-testauskehys, joka on suunniteltu yksikkö- ja integraatiotestaustyökaluksi, toimien selaimen kautta. Mocha mahdollistaa asynkronisten testien ajamisen ja testikattavuusraportoinnin sekä mockauksen, jolla voidaan välttää verkon hitautta. Jestin kasvava suosio on kuitenkin vähentänyt Mochan käyttöä. Mochan konfiguraatiot ovat erittäin muokattavissa, mutta tämä voi tehdä sen käyttöönotosta ja käytöstä monimutkaista. [48][49][50]

4.1.4. Jest

Jest on Facebookin kehittämä avoimen lähdekoodin testauskehys, joka alun perin kehitettiin React-projektien testaamiseen. Sittemmin sen käyttö on laajentunut käyttöliittymien testauksesta myös palvelinsovellusten testaamiseen, mutta sen pääfokus on edelleen front-end applikaatioiden koodin tarkastelussa. Jest tukee Node-, Typescript-, Vue- ja React -projektien testaamista ja se luokitellaan ns. yksikkötestaustyökaluksi, koska sitä pääasiallisesti käytetään sovelluksen koodin oikeellisuuden testaamiseen, vaikka sillä voi tehdä myös päästä päähän (end-to-end) -testejä. Jest on avoimen lähdekoodin testauskehysten joukossa yksi suosituimmista työkaluista, mikä johtuu sen helppokäyttöisyydestä ja tuesta monille eri teknologioille. [51][52]

Jest erottuu helppokäyttöisyydellään, sillä se tarjoaa valmiiksi käyttöön kykenevän out-of-the-box tyyppisen testauskehyksen. Asennus ei vaadi suurta asetusten määrittelyä [49]. Jest pystyy suorittamaan testejä yhdenaikaisesti, mutta valitettavasti siitä puuttuu asynkronisen testauksen tuki. Jest on kuitenkin nopeampi kuin esimerkiksi MochaJS ja tukee snapshot testausta, jolla se tunnistaa sisään tulevat muutokset ja ajaa testit automaattisesti jokaisen koodimuutoksen ilmaantuessa. Työkalu tukee myös mockausta ja koodikattavuuden analysointia. [50][51][52]

Jest on saavuttanut suosiota myös JavaScript-pohjaisten testauskehysten käytössä. Sen helppokäyttöisyys on selvästi lisännyt suosiota automaatiotestausalueella. [48]

4.2. Staattiset koodin analysointityökalut

Staattiset koodin analysointityökalut suorittavat automaattisesti koodille ja projektille skannauksia, joiden avulla etsitään ohjelmiston koodista huonosti tehtyjä osia tai riskialttiita riippuvuuksia. Nämä työkalut eivät vaadi testaajalta testitapausten implementointia, vaan ne suorittavat ominaisia skannauksia ja raportoivat havainnot testaajalle. Staattiset työkalut varmistavat, että koodimuutokset eivät sisällä haavoittuvuuksia, huonoa koodia tai virheitä [53]. Jatkuvan integraation ympäristöissä staattinen koodianalysointi auttaa havaitsemaan koodin heikkouksia kehityksen alkuvaiheessa.

Taulukko 3a. Staattiset testaustyökalut web-sovelluksille

Työkalu	**Avoimen lähdekoodin työkalu**	**Tuetut ohjelmointikielet**
SonarQube	Kyllä	C++, JavaScript, TypeScript, Python, Go, Swift, COBOL, Apex, PHP, Kotlin, Ruby, Scala, HTML, CSS, ABAP, Flex, Objective-C, PL/I, PL/SQL, RPG, T-SQL, VB.NET, VB6, XML + muut
Xray	Ei	C, C++, C#, Java, Groovy, Javascript, TypeScript, React, Vue.js, Objective-C 2.0, Scala, Python, Swift, Go, Visual Basic .Net, PHP, Ruby, Rational Software Architect models (C++), Kotlin, Clojure, Erlang, Apex
Fortify	Ei	ABAP/BSP, ActionScript, Apex, ASP.NET, C# (.NET), C/C++, Classic, ASP (with VBScript), COBOL, ColdFusion CFML, Go, HTML, Java (including Android), JavaScript/ AJAX, JSP, Kotlin, MXML (Flex), Objective C/C++, PHP, PL/SQL, Python, Ruby, Swift, T-SQL, VB.NET, VBScript, Visual Basic, XML, JSON/YAML, Terraform HCL and Docker (Dockerfile).

Taulukko 3b. Staattisten testaustyökalujen ominaisuudet

Työkalu	**Staattinen/dynaaminen**	**Käyttötarkoitus**	**Soveltuu kehitysputkessa testaamiseen**
SonarQube	Staattinen	Koodaustapojen	Kyllä
Xray	Staattinen	Mahdollisia	Kyllä
Fortify	Staattinen	Tietoturva	Kyllä

4.2.1. SonarQube

SonarQube on yksi suosituimmista koodianalyysityökaluista jatkuvan kehityksen ympäristöissä [53]. Se on itseohjautuva ja automaattinen koodintarkistustyökalu, joka auttaa järjestelmällisesti toimittamaan laadukasta koodia organisaation sääntöjen mukaisesti. SonarQube tuottaa laadun mittauksen, yleisten koodausstandardien mukaisten poikkeamien raportin sekä yksikkötestien kattavuuden raportin. Työkalu voidaan integroida kehitysputkeen, jolloin komponenttia voidaan testata jatkuvasti

automaattisesti. SonarQube tukee lähes 30 eri ohjelmointikieltä ja sen tarkistukset varmistavat koodin laatustandardien noudattamisen. [54][55]

SonarScannerin yksi tärkeimmistä toiminnoista on yksikkötestien kattavuuden raportointi koodiin tehtynä. Sen avulla SonarQube saa raportin kolmannen osapuolen tekemästä kattavuustestauksesta. Kun SonarQube on integroitu kehitysputkeen, analyysi voidaan asettaa vaatimaan yksikkötestien kattavuuden ylläpitoa ennen kuin koodimuutos hyväksytään osaksi päähaaraa. [55]

Java-implementaatio mahdollistaa SonarQuben integroimisen kehitysputkeen. SonarQube integroituu versionhallintaan, ja muutokset triggeröivät analyysityökalun ajamaan skannausta [55]. SonarQube luokittelee ongelmat tyypin ja vakavuuden mukaan. Virhe ilmenee, jos koodissa havaitaan vääränlainen implementaatio. Haavoittuvuus nostetaan, jos koodissa havaitaan kohta, joka voisi vahingoittaa järjestelmää. SonarQube voidaan konfiguroida organisaation tarpeiden mukaan raportoimaan koodin sisältämistä ongelmista, kuten huonosti koodatusta osasta tai muista ongelmista. [53]

4.2.2. Xray

Xray on staattinen koodianalysointityökalu, joka tarkistaa koodin käyttämät komponentit ja riippuvuudet mahdollisten haavoittuvuuksien varalta. Työkalu saa tiedon kehitysputki palvelimella ajetun koodin käännösprosessin alkamisesta, jonka jälkeen se skannaa toimintavaiheessaan kaikki riippuvuudet läpi ja raportoi haavoittuvuudet takaisin kehitysputki palvelimelle. Haavoittuvuuksia haetaan keskitetystä haavoittuvuuspankista, kuten esimerkiksi CVE. Jos haavoittuvuuksia löytyy, kehitysputki voidaan pysäyttää estämään niiden eteneminen. [56][57]

Xrayn skannauksen ajamista kehitysputken varhaisessa vaiheessa voidaan pitää matalan tason tietoturvatestinä. Se auttaa ehkäisemään tietoturvahaavoittuvuuksia ennen kuin koodia jaetaan eteenpäin, mikä helpottaa varsinaisen tietoturvatestauksen työtä ja vähentää riskiä haavoittuvuuksien pääsemisestä etenemään. Xray on ns. SCA-tyyppinen testaustyökalu. [57]

4.2.3. Fortify

Fortify on staattinen koodin analyysityökalu, joka keskittyy erityisesti tietoturvahaavoittuvuuksien etsintään. Työkalu käyttää useita algoritmejä ja laajenevaa tietokantaa ohjelmiston lähdekoodin analysointiin. Fortify on SCA-tyyppinen testaustyökalu, joka voi integroitua osaksi kehitysputkea ja skannata koodin haavoittuvuuksia automaattisesti. Työkalu tukee monia eri ohjelmointikieliä, kuten esimerkiksi Java, Python, C++ ja C#. [58]

4.3. Testiautomaatiokehykset käyttöliittymälle

Web-sovellusten automaatiotestauksessa käytetään työkaluja, jotka pystyvät suorittamaan käyttäjämäisiä toimintoja, kuten käyttöliittymän toiminnallisuuksien testaamista. Testaustapauksia voidaan rakentaa oletettujen toimintojen mukaisesti, jolloin työkalu lukee sovelluksen lähdekoodia ja toimii sen avulla suorittaen vaadittuja toimintoja. Lisäksi testitapauksiin voidaan yhdistää sovellukselle tarpeellisia

integraatioita, esimerkiksi lähettämällä sille toisen käyttöjärjestelmän lähettämää dataa tai toteuttamalla yhteisiä toimintoja. Testiautomaatiota voidaan hyödyntää sovelluksen hyväksymistestauksessa ja regressiotestauksessa, mikä keventää manuaalisen työn määrää ja nopeuttaa testausta. Kuitenkin testiautomaation kehittäminen ja ylläpito vievät aikaa, joten sen hyödyt tulee punnita suhteessa ajankäyttöön.

Käyttöliittymälle tehtävät automaatiotestit voidaan myös integroida kehitysputkeen, jolloin testit ajetaan sen jälkeen, kun sovellus on asennettu tarkoitettuun testiympäristöön. Uusien toiminnallisuuksien testaaminen edellyttää kuitenkin aina sovelluksen asennusta, mikä lisää kehitysputken läpimenoaikaa. Testien ajaminen on myös aikaa vievää, joten niiden määrää kannattaa harkita tarkasti suhteessa kehitysputken nopeuteen. Tässä tutkielmassa esitellään muutama yleisimmin käytössä oleva testiautomaatiokehys, joita käytetään nykypäivänä modernin web-sovelluskehityksen testaamisessa.

Taulukko 4a. Testiautomaatiokehyksiä selainpohjaisen sovelluksen testaamiseen

Työkalu	**Avoimen lähdekoodin työkalu**	**Ohjelmointikielet**
Selenium	Kyllä	Java, Javascript, Python, Ruby, Kotlin, C#
Robot Framework	Kyllä	Python+Robot
JBehave	Kyllä	Java
Cypress JS	Kyllä	Javascript

Taulukko 4b. Testiautomaatiokehysten ominaisuudet

Työkalu	**Käyttää seleniumia**	**Avainpohjainen testauskehys**	**Soveltuu kehitysputkessa testaamiseen**
Selenium	-	Ei	Kyllä
Robot Framework	Kyllä	Kyllä	Kyllä
JBehave	Kyllä	Kyllä	Kyllä
Cypress JS	Ei	Ei	Kyllä

4.3.1. Selenium

Selenium on avoimen lähdekoodin testauskehys, joka mahdollistaa selaintestaamisen ja sitä käytetään yleisimmin tähän tarkoitukseen [30]. Se toimii monilla eri koodikielillä, kuten Java, Python, Csharp, Ruby, JavaScript ja Kotlin, ja sen ytimenä toimii WebDriver, joka on yhteensopiva useiden selainten kanssa. Kun kaikki tarvittavat komponentit on asennettu, Selenium mahdollistaa testiautomaation toteuttamisen muutamalla koodirivillä. [59]

4.3.2. Robot Framework

Robot Framework on avoimen lähdekoodin automaatiokehys, joka on kehitetty Pythonilla, mutta sen käyttöä voidaan laajentaa muiden ohjelmointikielten kirjastoilla [60]. Useat alaa johtavat yritykset käyttävät Robot Frameworkia ja tukevat sen kehitystä [61]. Robot Framework on avainsanoihin perustuva automaatiokehys, joka soveltuu hyväksymistestaukseen, hyväksyntätestilähtöiseen kehitykseen, käyttäytymiseen perustuvaan kehitykseen ja robottiprosessiautomaatioon. Kehyksen käyttö on mahdollista hajautetuissa, heterogeenisissa ympäristöissä, joissa tarvitaan erilaisten teknologioiden ja rajapintojen käyttöä. Robot Frameworkin syntaksi on ihmisen luettavampaa, ja omien avainsanojen luominen on helppoa olemassa olevia avainsanoja hyödyntämällä. Koodipohjaisten komentojen toteuttaminen on myös mahdollista. Robot Frameworkin selainpohjaiset testit vaativat Seleniumin käyttöä, sillä Python edellyttää Seleniumin käyttöä selaimen käytön mahdollistamiseksi. [62]

4.3.3. JBehave

JBehave on Selenium-pohjainen testiautomaatiokehys, joka on kehitetty Javalla [59]. Kehys tukee käyttäytymiseen perustuvaa kehitystä, jossa käyttäytymistä kuvataan spesifikaatioina ja testitapauksina [63]. JBehaven moduulien kehittämiseen tarvitaan JDK 11, mutta testattavien projektien versioksi riittää JDK 8 [63]. Kuten Robot Frameworkissa, myös JBehave vaatii Seleniumin käyttöä selainpohjaisten testien suorittamiseen.

4.3.4. Cypress JS

Cypress JS on moderni web-käyttöliittymien testaustyökalu, joka mahdollistaa selainpohjaisten sovellusten testaamisen [64][65]. Työkalun tarkoituksena on helpottaa nykyaikaisten web-sovellusten automaatiotestauksen kehittämistä ja tehdä asiat mutkattomiksi kyseisiä käyttötarpeita ajatellen. Cypress on työpöytäsovellus, joka toimii MacOS-, Linux- sekä Windows-käyttöjärjestelmissä. Asennus edellyttää Node.js 12:n tai uudemman version käyttöä [63]. Testin käynnistyessä sovelluksen käyttöliittymä yhdistyy selainnäkymään, ja testiajon aikana syntyvät tapahtumat näkyvät samalla ruudulla [65]. Cypress ei vaadi Seleniumin WebDriverin käyttöä, kuten useat muut testiautomaatiokehykset. Se pystyy silti ajamaan testejä myös ulkoisissa selaimissa, mutta pääasiassa tuki rajoittuu Chromeen ja Firefox:iin [30]. Halutun selaimen voi antaa parametrina Cypressille, ja työkalu tukee kaikkien palveluntarjoajien kehitysputki-palvelimia. [65]

4.4. Kuormitustestaustyökalut

Kuormitustestauksen tarkoituksena on paljastaa sovelluksen suorituskykyyn liittyviä ongelmia [31]. Kuormitustestaukseen tarkoitetut työkalut mahdollistavat järjestelmän kuormituksen simuloimisen, jolloin systeemin käyttäytymistä ja suorituskykyä voidaan arvioida sille määriteltyjen vaatimusten puitteissa. Työkalujen ominaisuudet vaihtelevat tukemien komponenttien määrässä, ohjelmointikielissä, teknisessä suorituskyvyssä ja käytettävyydessä. Työkalun valintaan vaikuttaa itse testattavan

ohjelmiston luonne ja sen vaatimusmäärittelyt. Lisäksi eri työkalujen ominaisuudet rajaavat niiden käyttömahdollisuuksia erilaisten ohjelmistojen testauksessa. Esimerkiksi web-pohjaisten sovellusten testaamiseen tarvitaan yleensä tuki selaimille, jos sovelluksella on käyttöliittymällä suoritettavia toiminnallisuuksia. Työkalun kyky suoriutua useammasta tehtävästä helpottaa testaajan työtä ja välttää usean työkalun käyttämistä.

Taulukko 5a. Kuormitustestaustyökalut

Työkalu	**Avoimen lähdekoodin työkalu**	**Ohjelmointikielet**
Loadrunner	Ei	VB, VBscript, Java, JavaScript, C#
Locust.io	Kyllä	Python
Jmeter	Kyllä	Java
Blazemeter	Ei	Java
Webload	Kyllä	Javascript

Taulukko 5b. Kuormitustestaustyökalujen ominaisuudet

Työkalu	**Tuetut järjestelmät**	**Selain tuki**	**Soveltuu kehitysputkessa testaamiseen**
Loadrunner	Linux, Windows	Kyllä (Selenium)	Kyllä
Locust.io	Linux, Windows	Ei suoranaisesti	Kyllä
Jmeter	Linux, Windows, Mac	Ei suoranaisesti	Kyllä
Blazemeter	Selain-pohjainen	Käytetään selaimella	Kyllä
Webload	Linux, Windows	Kyllä (Selenium)	Kyllä

4.4.1. Loadrunner

LoadRunner on Hewlett-Packardin kehittämä automaattinen kuormitustestaustyökalu, jonka avulla testataan ohjelmiston käyttäytymistä ja suorituskykyä kuorman alla [66]. LoadRunnerin työkaluja käytetään web-applikaatioiden testauksessa ja niiden avulla voidaan kuvata koko järjestelmän läpäisevää (”end-to-end-”) suorituskykyä ohjelmiston hyväksymistestauksen aikana. Työkalun avulla voidaan vertailla saatuja tuloksia koodimuutosten välillä ja nähdä muutosten vaikutukset ohjelmiston suorituskykyyn. LoadRunner luo kuorman virtuaalikäyttäjien avulla, simuloiden tuhansien reaaliaikaisten käyttäjien kuormaa testattavan applikaation toimintoihin. Työkalu sisältää komponentteinaan virtuaalikäyttäjien generaattorin, kontrollerin, analyysit ja kuormitustestien generaattorit. LoadRunner tukee kaikkia selaimia, useita eri internet-sovelluksia ja koodikieliä, kuten VB, VBscript, Java, JavaScript ja C#. Työkalu toimii Windows- ja Linux-käyttöjärjestelmissä ja tarjoaa integraatiotuen useammille kehitysputki-palvelimille ja monitorointityökaluille. [66][67]

4.4.2. Jmeter

Apache Jmeter on avoimen lähdekoodin java-pohjainen applikaatio, jonka alun perin suunniteltiin web-sovellusten kuormitustestaukseen. Tällä hetkellä se tukee myös muita testausfunktioita, kuten Web – HTTP, HTTPS, SOAP, FTP, tietokanta via JDBC, LDAP, MOM via JMS, sähköpostit – SMTP, POP3 ja IMAP, natiiveja komentoja, skriptejä, TCP sekä Java objekteja. [69]

Jmeter tarjoaa käyttöliittymän, jonka kautta onnistuu testien suunnittelu, nauhoittaminen, testien ajaminen sekä virheiden korjaaminen. Testitulokset voidaan esittää helposti luettavissa olevissa muodoissa, kuten HTML, JSON, XML tai tekstimuotoisena. [69]

Jmeter toimii protokollatasolla ja sen vahvin ominaisuus on hajautettu testaus [67]. Vaikka työkalu näyttää web-selaimelta ja antaa tehdä toimintoja selainmaisesti, se ei tue kaikkea selainpohjaista testausta eikä toimi selaimen kautta löytyvien HTML-elementtien avulla. Toimintojen tekeminen tapahtuu rest-kutsuina kohdekomponenttia kohti. Siksi Jmeter sopii paremmin rest-pohjaisten sovellusten tai rajapintojen testaamiseen kuin käyttöliittymän testaamiseen. [67]

4.4.3. Blazemeter

Blazemeter on pilvipohjainen työkalu, jonka toiminnallisuudet perustuvat pitkälti Jmeterin ominaisuuksiin. Työkalu tukee suoraan Jmeter-skriptien ajamista ja testien muokkaamista Blazemeterin oman käyttöliittymän kautta. Pilvipohjaisuus mahdollistaa toimintojen suorittamisen missä tahansa sijainnissa selaimen avulla, ja skriptien tallentaminen onnistuu chrome-laajennuksen avulla. [69]

Lisäksi Blazemeterissä on mahdollista hallinnoida muita testausvaiheiden testejä, kuten funktionaalisia käyttöliittymätestejä. Työkalu on myös integroitavissa kehitysputkeen, esimerkiksi Jenkinsiin tarkoitetun plugarin tai Azure DevOps -kehitysputkeen lisätyn laajennuksen avulla. [69]

4.4.4. Locust.io

Locust.io on avoimen lähdekoodin kuormitustestaustyökalu, joka on pääosin suunniteltu web-ohjelmistojen testaukseen, mutta sitä voidaan käyttää myös minkä tahansa systeemin tai protokollan testaamiseen. Työkalua voi käyttää web-käyttöliittymän avulla tai ilman, ja testit toimivat tapahtumapohjaisesti, mikä mahdollistaa suurten kuormien ajamisen yhdeltä prosessorilta. [70]

Locustin lähdekoodi on pidetty tarkoituksella pienenä, jotta se olisi helposti adaptoitavissa erilaisiin tilanteisiin [70]. Kun samaa työkalua voidaan käyttää useampaan eri osaan systeemiä sen suorituskyvyn testatessa, testauksen suorittaminen helpottuu. Toisaalta Locust häviää esimerkiksi Jmeterille, koska se tukee vain HTTP-web-pohjaista testausta. Uusien toimintojen lisääminen vaatii usein lisätyömäärää ja osaamista testausta suunnittelevan tahon puolelta, vaikka työkaluun on mahdollista lisätä pythonin tukemia ominaisuuksia lisäämällä python-funktioita. [71]

4.4.5. Webload

Webload on vaihtoehto Loadrunnerille, joka sopii erityisesti monimutkaisten toiminnallisuuksien sekä korkean kuorman testaamiseen. Työkalun koodikielenä toimii natiivi Javascript, joka mahdollistaa kompleksisemman bisneslogiikan ja funktiokirjastojen tuen. [72]

Webload tukee myös Seleniumia ja Perfecto Mobilea, joiden avulla on mahdollista mitata oikeaa käyttöliittymäkokemusta web- ja mobiilikäyttöliittymissä. Kuorman generointi on mahdollista pilvipalveluista, kuten AWS:stä, sekä fyysisistä koneista. Työkalu tarjoaa käyttäjälle käyttöliittymän, jonka avulla käyttäjä voi luoda räätälöityjä raportteja ja seurata testituloksia reaaliajassa. [72]

Lisäksi Webloadille on tarjolla Jenkins-laajennus, joka mahdollistaa työkalun käytön jatkuvan toimittamisen prosesseissa. [72]

4.4.6. Dynatrace

Dynatrace on ohjelmistotiedon seuranta-alusta, joka tarjoaa käyttäjälle tietoa sovellusten suorituskyvystä ja infrastruktuurista [73]. Vuonna 2022 Dynatrace oli johtava infrastruktuurin monitorointityökalu IT-alalla. Dynatrace auttaa käyttäjää havainnoimaan sovelluksen käyttäytymistä ja suoriutumista sen ollessa käytössä tai testauksen alla. Työkalun automaattinen kooditason näkyvyys ja juuriongelmien havaitsemisen kyky mahdollistavat kompleksienkin ympäristöjen seurannan [74].

Dynatrace soveltuu hyvin myös suorituskykytestaukseen ja auttaa analysoimaan järjestelmän toimivuutta. Työkalua voi käyttää myös kuormitus- ja poikkeustilannetestausten suorittamiseen applikaatioille. Useammat suoritustestaustyökalut ovat integroitavissa Dynatraceen, mikä mahdollistaa testaustulosten kattavamman raportoinnin ja analysoinnin.

5. WEB-SOVELLUKSEN KEHITYSPUTKI

Tässä tutkimuksessa mukana olevan projektin kehitys tapahtuu kehitysputken menetelmien mukaan. Projektin integraatioputkena toimii katselmus- ja toimitusputki, jotka siirtävät koodimuutokset ohjelmistokehittäjän työpöydältä kohti julkaisuputkea. Katselmointi- ja toimitusputket noudattavat kehitysputki menetelmiä, jotka on esitetty luvussa 2. Ne suorittavat kehitystiimin sisäisessä kehitysputkessa tapahtuvat toimenpiteet ja toimittavat sovellusversioita kehitystiimiltä eteenpäin. Automaatiopalvelimena käytetään Jenkinsiä, joka on laajasti käytössä oleva avoimen lähdekoodin palvelu.

Tutkielmassa kuvataan web-sovelluskehitykseen soveltuva ohjelmistokehitysputken malli, jonka avulla pyritään varmistamaan sovelluksen toimivuus mahdollisimman varhaisessa vaiheessa sovelluskehitystä. Tavoitteena on tuoda automaatiotestaus lähemmäksi ohjelmistokehitystä, jolloin mahdolliset laatua heikentävät tekijät voidaan ehkäistä jo ohjelmistotuotannon alkuvaiheessa.

5.1. Kehitysmenetelmien muutostarpeen tunnistaminen

Tutkimuksen aikana kehitystiimin kanssa järjestettiin työpaja, jossa arvioitiin nykyistä tilannetta ja tavoitetilaa testiautomaation ja kehityksen aikana tehtävissä toimenpiteissä. Lähtötilanteessa testiautomaatiota suoritettiin vain yksikkötestauksen tasolla ennen koodikatselmointia, ja GUI-automaatiotestejä (”GUI”, graphical user interface) ajettiin vasta uusien toiminnallisuuksien testaamisen yhteydessä, jolloin julkaisu oli jo jäänyt jälkeen. GUI-automaatiotestien seuranta ja ylläpito olivat testaajan vastuulla, mikä johti siihen, että kehittäjät eivät hyödyntäneet olemassa olevia testejä ohjelmistovirheiden havaitsemisessa. Testiautomaation ylläpito ja vastuu olivat myös yksittäisen testaajan vastuulla, mutta manuaalitestauksilta jäänyt aika ei riittänyt automaatiotestien kehittämiseen. Koodimuutokset integroitiin yleensä päähaaraan, ja kehittäjä alkoi keskittyä uuden toiminnallisuuden toteuttamiseen ennen kuin edellinen muutos oli varmistettu automaatio- tai manuaalisesti tehdyillä testeillä. Kun automaatiotestaus oli kattavaa, sen mahdollisuus havaita virheitä kasvoi ja samalla se testasi jo etukäteen testaajan manuaalitesteissä tekemiä testejä.

Kehitystiimin työpajan tuloksena päädyttiin kokeilemaan GUI-testiautomaation suorittamista jo ennen koodikatselmointia, jotta kehittäjät näkisivät heti muutosten aiheuttamat rikkoutuvat testit. Tämän myötä testiautomaation kehitys- ja ylläpitovastuuta voitaisiin jakaa useamman henkilön kesken. Verifiointi tapahtuisi testiautomaation avulla ennen koodikatselmointia ja hyväksytty automaatiotulos olisi lisävarmiste koodin yhdistämiseen päähaaraan katselmoinnin jälkeen. Tämän ansiosta päähaaraan ei yhdistettäisi muutoksia, joita ei ole vielä verifioitu automaatiotestien avulla.

Tiimissä päätettiin kokeilla GUI-automaatiotestien ajamista ennen koodikatselmointia, mutta huomattiin, että se hidasti kehitysputken läpimenoaikaa. Silti automaatiotestien merkitys ohjelmistovian havaitsemisessa oli tärkeää. Yksi ratkaisu olisi pienentää testisettiä katselmointiputkessa, mutta tämä voisi rajoittaa mahdollisuuksia havaita kaiken tyyppisiä ongelmia. Loput testit voitaisiin ajaa toimitusputkessa koodihaarojen yhdistymisen jälkeen.

5.2. Tarvittavat toimenpiteet

Sovelluksen aikaisemmat automaatiotestit oli toteutettu Jbehave-testiautomaatiokehyksellä, mutta uudistuksen myötä käyttöliittymältä ajettavien testien tekeminen jouduttiin uusimaan kokonaan. Tutkimuksen toteuttamiseksi tarvittiin siis alusta loppuun implementoitu testiautomaatio käyttöliittymille. Tavoitteena oli kattaa lähes kaikki sovellukselle ominaiset käyttötapaukset automaatiotesteillä. Poikkeustilanteiden ja rajatapausten testaamiseen katsottiin manuaalitestauksen olevan nopeampi ja helpompi toteuttaa, eikä niiden takia tarvitsisi ylläpitää muuta kehitystä estäviä konfiguraatioita eri toiminnallisuuksien mahdollistamiseksi. Kattavan automaation ansiosta manuaalinen regressiotestaus voitaisiin suorittaa vasta koodihaarojen yhdistämisen jälkeen. Kuitenkin asennusvaiheen tutkivaa manuaalitestausta voidaan tarvittaessa suorittaa ennen koodihaarojen yhdistämistä, mikäli se on tarpeen.

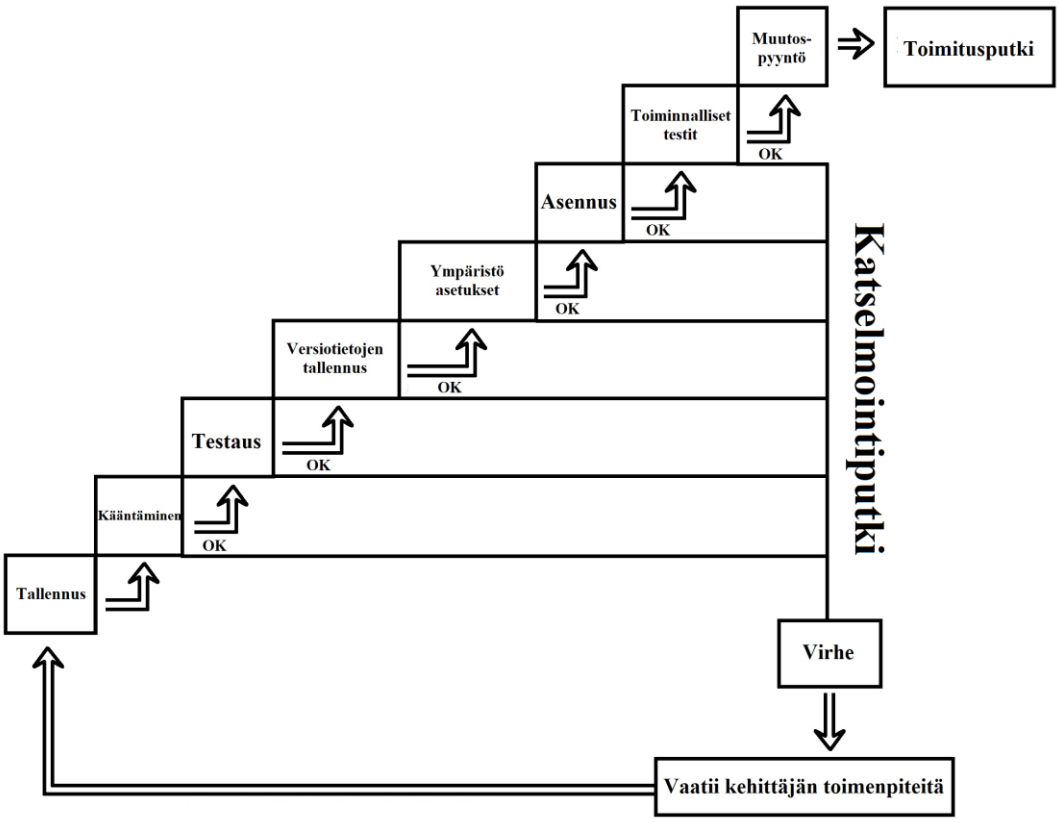
5.2.1. Testiautomaation toteutus Robot Frameworkilla

Sovelluksen GUI-testiautomaation toteutuksessa päätettiin käyttää Robot Framework -testiautomaatiokehystä, joka oli jo aiemmissa implementoinneissa osoittautunut hyväksi vaihtoehdoksi testiautomaatioiden tekoon. Aiemmin käytetty Jbehave-kehys koettiin haastavaksi ylläpitää ja kehittää, kun taas Robot Framework oli todettu helppokäyttöiseksi. Tämän vuoksi Robot Framework valittiin nopeimmaksi ratkaisuksi aikataulullisista syistä. Koska olin kehitystiimissä kokenein Robot Framework -testien kirjoittaja, minulle annettiin tehtäväksi testiautomaatioiden toteutus.

Testien kehittäminen voi olla haastavaa, kun sovelluksen toiminnallisuudet eivät ole vielä lopullisesti valmiita ja käyttöliittymä elää viimeisiin muutoksiin asti. Yleensä automaatiotestit rakennetaan vasta, kun sovellusmuutokset ovat stabiloituneet ja testejä tarvitsee tehdä vain kerran. Tässä tapauksessa sovelluksen toiminnallisuudet olivat jo lähes valmiita, joten testien implementointi onnistui sujuvasti. Testien toteutuksessa hyödynnettiin Robot Frameworkin avainsanapohjaista testiautomaatiokehyksen käyttöliittymää. Robot Frameworkilla käytetään valmiita kirjastoja ja uusia avainsanoja voidaan luoda sovelluksen tarpeiden mukaan.

Automaatiotestien kattavuus saatiin tutkimuksen aikana halutulle tasolle. Testien läpimenoaika vaihteli 7 ja 9 minuutin välillä riippuen testiympäristön kyvykkyydestä. Testien suoritus on nopeampaa lokaalisti kuin automaatiopalvelimen kautta, sillä Jenkins-palvelin tekee alustavat asennukset ja valmistelut jokaisen ajon aluksi ennen varsinaisten testien suorittamista. Pyrimme jatkuvasti kehittämään testien vakautta, jotta ne eivät turhaan hidasta kehitysputkea.

5.3. Katselmointiputki (Review-pipeline)



Kuva 4. Katselmointiputki.

Katselmointiputki on ensimmäinen automaatioputki, joka tarkastelee kehittäjän tekemiä muutoksia. Katselmointi perustuu neljän silmän periaatteeseen (four-eyes-principle) [94], joka edellyttää koodimuutoksen tarkistamista toisen henkilön toimesta. Kehitysputkessa tämä tarkoittaa, että automaation ja ohjelmistokehittäjän tulisi suorittaa tietyt toiminnot ennen toisen henkilön katselmointia. Tällä tavalla mahdolliset virheet voidaan havaita jo ennen katselmointia, mikä vähentää tarvetta useille katselmuksille ja testaajien ylimääräisille testauksille. Kuvassa 4 esitetään toimintojen eteneminen katselmointiputkessa.

5.3.1. Koodin kääntäminen (Build)

Kun versionhallinta havaitsee muutoksen, käynnistetään katselmointiputken ensimmäinen vaihe, joka on koodin kääntäminen. Kääntäminen tapahtuu käyttäen ohjelmistoprojektin hallintatyökalua Mavenia [75].

5.3.2. Yksikkötestaus (Test)

Mavenin suorittama testausvaihe sisältää yksikkötestien suorittamisen jUnitilla sekä staattisen koodianalyysin SonarQubella. Ennen koodianalyysia on hyvä suorittaa yksikkötestit, jotta mahdolliset virheet voidaan havaita ja välttyä turhalta skannaamiselta. SonarQube havaitsee standardien vastaisen koodin sekä logiikan puutteet projektin sisältä. Lisäksi suoritetaan tietoturvaskannaus riippuvuuksien

sisältämien haavoittuvuuksien varalta. Loput skannaukset suoritetaan vasta toimitusputkessa, jotta kehitysvaihe pysyy nopeana ja tehokkaana.

5.3.3. Versiotietojen tallennus (Publish artifact)

Tässä vaiheessa kehitysputkea koodin laatu on tarkistettu kooditasolla, ja jos kaikki edellä mainitut vaiheet on suoritettu onnistuneesti, ohjelmistosta voidaan luoda asennettava versioitu paketti. Tämä paketti voidaan tarvittaessa siirtää eri ympäristöihin testattavaksi. Katselmointiputken seuraavassa vaiheessa käytetään tätä pakettia sovelluksen asentamiseen, ja tarvittavat ympäristökohtaiset asetukset asetetaan valmiiksi.

5.3.4. Asennus (Deploy)

Käyttöliittymäsovelluksen muutosten testaaminen on mahdollista vasta, kun sovellus on asennettu johonkin ympäristöön, jossa sitä voidaan käyttää käyttöliittymältä käsin. Ympäristö määrittelee sovelluksen toimintaympäristön, jossa testaamalla voidaan varmistaa sen toiminnallisuus oikeiden käyttötapauksien kanssa.

Testaaminen tapahtuu testiympäristössä, joita tarvitaan useampia, jotta kehittäminen ja testaaminen voidaan suorittaa turvallisesti ennen tuotantoon vientiä. On tärkeää, että kehitystiimeillä on oma testausympäristönsä, jossa muiden tiimien kehitys tai verifiointi eivät häiritse kehitysputken suorittamia asennuksia. Asennukset voivat aiheuttaa käyttökatkoja sovelluksissa, jolloin sovelluksen käyttäjät joutuvat odottamaan niiden suorittamista.

Asennusvaihe kehitysputkessa on nyt valinnainen ja sen tarve määritellään kehitystiimin ehdolla. Esimerkiksi versionhallinnan ulkopuoliset muutokset, kuten automaatiotestit, eivät vaadi asennusta testiympäristöön, jolloin kehitysputkeen lisättävä ehto takaa lyhyen läpimenoajan. Kun taas kehittäjä tuo testattavia toiminnallisuuksia, asennus asetetaan suoritettavaksi, minkä jälkeen GUI-testit ajetaan asennetulle sovelluspaketille.

5.3.5. Käyttöliittymä-testiautomaatio

Käyttöliittymän testiautomaatiot ajetaan asennuksen jälkeen, ja niiden suorittaminen riippuu kehitysputkessa suoritettavan asennuksen ehdosta. Automaatiotestien on tärkeää olla nopeita ja vakaita, jotta niiden suorittaminen kehitysputkessa olisi sujuvaa ja aiheuttaisi mahdollisimman vähän häiriöitä.

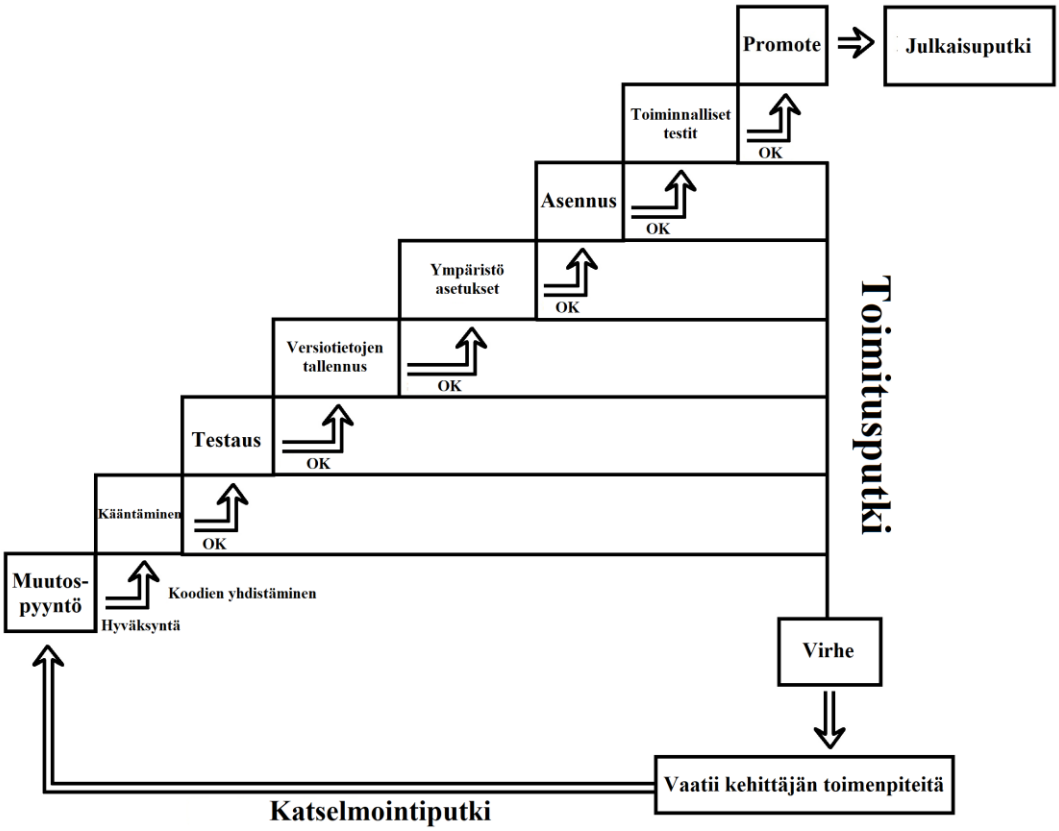
Testien läpimeno vie pisimmän ajan kaikista kehitysputken vaiheista, mutta niiden onnistunut suorittaminen vaatii kaikkien testien hyväksymisen. Tämän vuoksi testiautomaation suorittaminen kehitysputkessa vaati nopean epäonnistumisen alasajon, jotta testiajot eivät tukkisi kehitysputkea. Robot Frameworkilla löytyi --exitonfailure-komento [76], joka mahdollisti koko testiajon lopettamisen, jos yksikin testi epäonnistui.

Tutkimuksen tavoitteena oli saada kehittäjät osallistumaan testiautomaation ylläpitoon ja kehitykseen jo koodimuutosten yhteydessä. Tämä vaihe lisättiin katselmointiputkeen, jotta kehittäjät joutuvat tarkistamaan muutoksiensa vaikutukset automaatiotesteihin ja varmistamaan niiden suoriutumisen.

5.3.6. Muutospyyntö (Pull Request)

Kehittäjän koodimuutoksen tallennuksesta syntyy muutospyyntö (”PR”, pull request), joka vaatii onnistuneen katselmointiputken suoriutumisen ennen kuin koodimuutos voidaan yhdistää päähaaraan. Katselmoinnin päätepisteessä toinen kehittäjä tai testaaja hyväksyy muutoksen viemisen eteenpäin, kun koodimuutokset on katselmoitu ja koodikäännös onnistunut. Tämän jälkeen koodimuutos voidaan siirtää kehityksessä seuraavaan vaiheeseen.

5.4. Toimitusputki (Delivery-pipeline)



Kuva 5. Toimitusputki.

Kun koodikatselmointi on hyväksytty, koodimuutos voidaan yhdistää ohjelmiston päähaaraan, mikä käynnistää toimitusputken. Tämä toimitusputki paketoi muutoksen olemassa olevan koodin mukaan ja luo valmiin version sovelluksesta toimitusta varten.

Kuvassa 5 esitetään katselmointi- ja toimitusputken toiminta yhdessä. Toimitusputki eroaa katselmointiputkesta siinä, että se sisältää koodimuutokset myös muista kehittäjien haarautumista, jotka on yhdistetty yhteen päähaaraan. Lisäksi toimitusputkessa suoritetaan aiemmin jätetyt, enemmän aikaa vievät staattiset koodianalyysiskannaukset. Toimitusputken lopussa toiminnallisuudet menevät hyväksymistestaukseen, joten asennus- ja automaatiotestausvaiheet ovat nyt pakollisia vaiheita.

Toimitusputkessa asennus tehdään edelleen samaan testiympäristöön, kuin katselmointiputkessa tehty asennus. Ohjelmistokehittäjät voivat määrittää haluavatko käyttää samaa vai eri ympäristöä. Useamman ympäristön käyttö on yleensä parempi

ratkaisu, jos käyttäjiä on useita. Toimitusputkessa luodaan julkaisukelpoinen versio sovelluksesta, joka on paketoitu päähaaran päälle ja odottaa manuaalitestausta, kunhan kaikki vaiheet on suoritettu onnistuneesti.

5.4.1. Muutoksen esittäminen ylemmälle tasolle (Promote)

Toimitusputken viimeisessä vaiheessa koodimuutos esitetään ylemmälle tasolle vietäväksi, tätä vaihetta kutsutaan Promote-vaiheeksi. Promote-vaiheessa valmiiksi testattu sovellusversio ehdotetaan vietäväksi seuraavaan ympäristöön jatkotestauksia varten. Promote-vaihe lähettää pyynnön seuraavalle taholle uuden sovellusversion käyttöönotosta julkaisuputkessa. Sovelluspäivitykset integroituvat automaattisesti kehitystyön seurantajärjestelmään ja ovat hallittavissa kehitystiimin vastuulla olevien tehtävien statuksen avulla.

5.5. Julkaisuputki (Release-pipeline)

Julkaisuputki on tärkeä osa ohjelmistokehitystä ja se toimii yhtenä polkuna tiimin ja tuotannon välillä. Putki sisältää lopullisen hyväksymistestauksen ennen kuin komponentit viedään tuotantoon. Hyväksymistestausta suoritetaan useassa eri ympäristössä ja sen määrä riippuu komponenttimuutosten laajuudesta. Putkessa ajetaan sovelluksen automaatiotestit jokaisessa ympäristössä ja varmistetaan, että sovellukset toimivat yhdessä ja regression varalta.

Julkaisuputki on integroitu järjestelmään, joka ylläpitää tiimin kehitystyöjonoon asetettuja tehtäviä. Sovelluksen edistymistä voidaan seurata eri testausvaiheiden perusteella. Hyväksytysti putken läpi edennyt sovellus päätyy lopulta asiakkaiden käyttöön.

5.6. Muut testausvaiheet

Kehitystiimin vastuulla on varmistaa sovelluksen kuormituskyvykkyys ja tietoturva ennen tuotantoon vientiä. Testaukset suoritetaan keskitetyn kuormitustestaustiimin tai tietoturvatestaustiimin toimesta. Testien suorittaminen on hyvä tehdä mahdollisimman varhaisessa vaiheessa, jotta mahdolliset ongelmat huomataan ajoissa. Esimerkiksi kuormitustestaus kannattaa aloittaa heti, kun sovelluksen toiminnallisuudet mahdollistavat tutkivan kuormitustestauksen. Tietoturvatestaus ja auditointi puolestaan kannattaa aloittaa heti, kun sovellus täyttää suunnitelmissa laaditut tietoturvavaatimukset. Koodimuutosten yhteydessä kehitystiimin tulee analysoida kuormituksen ja tietoturvatestauksen tarpeellisuus.

5.7. Testaajan ja kehittäjän rooli ohjelmistotestauksessa

Kehitystiimin testaajan vastuulla on varmistaa tuotteen toiminnallinen laatu suunnittelemalla ja määrittelemällä tuotteelle testausvaiheet. Kehittäjien vastuulla on testien kehittäminen ja testaaminen kehitysputkessa. Testaajan tehtävä on varmistaa, että tuote on käynyt läpi kaikki testausvaiheet ja on toiminnallisesti laadukas. Testaajan työnkuva ei sisällä kooditason testaamista, vaan he keskittyvät sovelluksen toiminnallisuuksien testaamiseen.

Kehittäjän vastuulla on tuottaa koodimuutoksia ja huolehtia testiautomaatiosta. Testaaja toimii toiminnallisten testien kehittämisen tukena ja voi auttaa testiautomaation kattavuuden lisäämisessä monimutkaisempien testien avulla. Lisäksi testaaja voi auttaa testiautomaatiovelan poistamisessa. Kehittäjien ja testaajan vastuulla on jakaa käyttöliittymätestien kehittäminen tasaisesti, jotta testiautomaation rakentaminen ei ole pelkästään testaajan vastuulla.

Vanhenevien toiminnallisten testien ylläpito ja korjaaminen voi olla tarpeen myös ilman koodimuutoksia. Jatkuvan ylläpidon mahdollistaa kehittäjien tekemä muutosprosessi, mutta mikäli muutoksia ei tule, testien korjaus voidaan vastuuttaa myös testaajan tehtäviin. Kuitenkin testiautomaatio-osaaminen laajenee tiimin sisällä, kun testejä kehitetään useampien henkilöiden toimesta, ja näin kulloisenkin tehtävän suorittaminen voidaan vastuuttaa aikaa omaavalle henkilölle. Testiautomaation käyttäminen myös muissa kuin kehitystiimissä vaatii testien jatkuvaa seurantaa ja ylläpitoa. Muualta kuin kehitystiimistä tulevat toimenpiteet testien ylläpitoon jalkautuvat ensisijaisesti testausvastuussa olevan henkilön kautta.

6. POHDINTA

Tutkielmassa tutkittiin web-pohjaisen sovelluksen testaamista kehitystiimin kehitysputkessa. Yleensä automaatiotestien ylläpito ja kehittäminen on testaajien vastuulla, mikä voi johtaa siihen, että testien korjaaminen ja ylläpito jäävät kehityksen viimeisiin vaiheisiin. Tämä ei ole yhteensopiva testivetoisen kehityksen periaatteen kanssa, jossa testitapaukset luodaan ennen kehitystä ja koodimuutoksilla pyritään varmistamaan testitapauksien läpäisy [77]. Viime hetken testien tekeminen ei auta kehittäjää kehityksen aikana ja voi viivästyttää sovelluksen toimitusta, jos ongelmia löytyy vasta loppuvaiheessa. Tutkielmassa pyrittiin löytämään ratkaisu siihen, mitä asioita tulisi suorittaa automaation ja ohjelmistokehittäjän toimesta jo ennen toisen henkilön katselmointia. Tutkielmassa myös pohdittiin, voitaisiinko kehitysputkessa tehtävien muutosten avulla jakaa testiautomaation kehittämis- ja ylläpitovastuuta paremmin tiimijäsenien kesken. Tavoitteena oli saada automaatiotestien kehittäminen ja ylläpito jatkuvaksi prosessiksi sovelluskehityksen aikana.

Tutkielmassa kokeiltiin uutta työskentelymallia web-pohjaisen sovelluksen testaamisessa kehitystiimin kehitysputkessa. Muutokset otettiin käyttöön yhden projektin aikana, ja projektin seurannan avulla tarkoituksena oli selvittää, toimiiko uusi työskentelymalli testiautomaation ylläpidon ja vakauden kannalta tehokkaammin. Tutkimuksen aikana havaittiin, että kehittäjien kiinnostus käyttöliittymätestien vakauteen ja toimivuuteen kasvoi. Kehittäjät tarkastivat itse testiautomaation tuloksia ja varmistivat koodin toimivuuden ennen muutoksen viemistä eteenpäin, kun katselmointiputki pysähtyi robot-testeihin. Testaajan tuki säilyi vahvasti läsnä testitulosten analysoinnissa sekä testien korjauksessa.

Testien ajaminen lisääntyi merkittävästi, kun testit ajettiin jokaiselle asennukselle ja ajoja tehtiin useasti päivässä lähes jokaiselle koodimuutokselle. Testitulosten analysoinnin määrä kasvoi myös huomattavasti, kun koodimuutosten läpivieminen edellytti myös jatkuvaa testitulosten seurantaa. Haasteena koettiin koodimuutosten hidastuminen, jos käyttöliittymätesteissä havaittiin ongelmia. Projektin suunnitelmana on jatkaa uusien työskentelytapojen käyttöä ja seurata niiden toimivuutta pitkällä aikavälillä. Mikäli uudet toimintatavat osoittautuvat toimiviksi, niitä voidaan ottaa käyttöön myös tiimin muihin ylläpitämiin ja kehittämiin ohjelmistoihin. Kehittäjien tulee myös itse kehittää automaatiotestejä omien koodimuutostensa verifioimiseksi. Tulevaisuudessa voitaisiin tutkia tapoja, joilla uusia käytäntöjä voitaisiin jalkauttaa laajemmin koko organisaatiossa.

Kehitysputkessa testiautomaatioiden vakaus on ratkaisevassa roolissa, sillä sen varaan on asetettu hyväksytyn läpimenon ehto. Laajamittainen ja luotettava verifiointi on tärkeää toiminnallisuuksien testaamisessa. Epävakaus kehitysputkessa vaikuttaa negatiivisesti kehityksen etenemiseen ja kehittäjien ilmapiiriin. Vakauteen vaikuttavat myös tiimistä riippumattomat ongelmat, kuten eri järjestelmien yhteysongelmat ja ajoympäristön seikat.

Testauksen kehitys kohti automaatiota jatkuu, mutta manuaalitestaukselle on edelleen paikkansa. Manuaalitestaus on tärkeää varsinkin tapauksissa, joissa automatisointi ei ole mahdollista tai testitapaukset ovat liian monimutkaisia automatisoitavaksi. Kehitystiimin tulee kuitenkin pyrkiä minimoimaan manuaalisesti tehtävän testauksen määrää. GUI-automaatiotesteissä tulee arvioida automatisoitavien testitapausten määrää ja uusien ominaisuuksien kehityksessä tulee huomioida mahdollisten virhelöydösten automatisointi. Automaatiotasoa voidaan nostaa

lisäämällä automaatiotestejä havaittuihin virheisiin. Tämä helpottaa tulevaisuudessa mahdollisten virheiden havaitsemista jo varhaisessa vaiheessa.

Ohjelmiston testaaminen on tärkeää, jotta lopputuote toimii sujuvasti todellisessa käyttöympäristössä. Kattavan testauksen saavuttamiseksi käytetään eri testausprosesseja, kuten testiautomaatiota ja manuaalitestausta. Nämä prosessit suoritetaan pitkin sovelluksen elinkaarta, jotta voidaan varmistaa ohjelmiston toimivuus ja välttää ongelmia tuotteen käytössä.

Tutkielman testausprosesseja tarkastelleessa osiossa käsiteltiin monipuolisia testaustyökaluja, joilla voidaan toteuttaa automaatiotestausta. Lisäksi tarkasteltiin kuormitustestaustyökaluja, joita voidaan käyttää lisäämään testien monipuolisuutta kehitysputkessa. Tutkielmassa käsitellyn sovelluksen kehitysputkeen ei sisällytetty kuormitustestausta, koska sen suorittamiseen käytettävä aika ja sen vaikutus sovelluksen kapasiteettiin oli suhteellisen pieni. Kuitenkin kapasiteettiin vaikuttavien muutosten verifioimiseksi on suositeltavaa lisätä kuormitustestaus kehitysputkeen projekteissa, joissa se on tarpeellista. Tällaisissa projekteissa kapasiteettiin vaikuttavat muutokset tulisi testata välittömästi muutosten syntymisen jälkeen, jotta läpimenoaika pysyy nopeana.

Kehitysputkessa käytetyt testaustyökalut syventävät testauksen tasoa huomattavasti verrattuna pelkästään ihmisen avulla suoritettaviin testeihin. Lisäksi testaaminen kehitysputken sisällä nopeuttaa palautteen saamista koodimuutoksista toimintavarmuuden varmistamiseksi, kun taas testaaminen vasta valmiilla sovelluspaketeilla hidastaa palautteen saamista. Työkalut mahdollistavat monipuolisen tarkastelun kooditason laatuun, koodaustapoihin ja käytettäviin komponentteihin. Käyttöliittymätason automaatiotestit nopeuttavat toiminnallisten testien suorittamista ja vapauttavat kehitystiimille aikaa uusien toiminnallisuuksien kehittämiseen. On kuitenkin muistettava, että jokaisen työkalun käyttö ei ole yksinkertaista ja koodianalyysit voivat löytää ongelmia myös oikein tehdyistä muutoksista. Toiminnalliset testit ovat riippuvaisia testiympäristön stabiiliudesta sekä suoritettavien testien laadusta, ja testattaessa on otettava huomioon testattavan järjestelmän toimintaympäristö. Erityisen tärkeää tämä on pankkisovelluksen testaamisessa, jossa testit ovat kriittinen osa ohjelmistokehitystä.

Tulevaisuudessa pyritään yhä enemmän nopeisiin ja luotettaviin toimituksiin, joissa myös testaus tapahtuu automaattisesti. Tämä edellyttää suoritettavien testien kattavuuden ja monipuolisuuden saamista mahdollisimman korkealle. Luotettavat ja vakaat testit ovat välttämättömiä, jotta ne eivät viivästyttäisi toimitusta. Testiympäristöjen on toimittava vakaasti ilman ylimääräisiä esteitä sovelluksen toiminnalle tai testaamiselle. Vaikka manuaalista työtä ei voida korvata kokonaan koneellisesti, automaatiotestaus nopeuttaa kokonaistyötä huomattavasti ja toimii hyvänä tukena manuaaliselle testaukselle. Tutkivan testauksen ja uusien toiminnallisuuksien testauksessa manuaalitestausta tulee käyttää ensisijaisena testaustapana, jotta muutoksien toimivuudesta saadaan nopeasti palautetta. Toisaalta regressiotestauksessa ja laajojen kokonaisuuksien testaamisessa automatisoidut testit ovat manuaalista testausta tehokkaampia.

Käyttöliittymätestausten automatisointi tulee jakaa sopivasti kehittäjien ja testaajien välillä. Kehittäjien vastuulla on yksikkötestien ja koodianalyysityökalujen käyttö jo kehitysvaiheessa. Funktionaalisten käyttöliittymätestien tekeminen on mahdollista vasta muutosten valmistuttua, jolloin kehittäjät voivat toteuttaa automaatiotestit toteutetulle muutokselle ennen sen viemistä ohjelmiston päähaaraan. Kuitenkin kaikkien käyttöliittymätestien tekemistä ei kannata jättää kehittäjien

vastuulle. Neljän silmän periaatetta mukaillen on hyvä, että testejä lisätään myös toisen henkilön toimesta. Testaajan vastuulle jää automaatiotestien lisääminen sellaisille testitapauksille, jotka jäävät käyttötapausten ulkopuolelle. Testaajan on myös vastattava testikattavuuden parantamisesta ja regressioiden ehkäisemisestä testiautomaation ja manuaalitestauksen avulla.

7. YHTEENVETO

Tutkielmassa tutkittiin keinoja estää testiautomaation ylläpidon ja kehityksen eriytymistä. Tutkimuksen kohteena oli web-pohjainen pankkisovellus ja sen kehitystiimin kehitysputki, johon kuului useita automaatiotestauksen vaiheita. Automaatiotestaus vähentää manuaalisen testauksen tarvetta ja mahdollistaa testaajille enemmän tutkimustyötä. Lisäksi automaation käyttö jo kehitysvaiheen alkuvaiheessa pienentää riskiä virheellisen koodin toimittamiselle ja vähentää virheistä johtuvia viivästyksiä.

Tutkielmassa perehdyttiin työkaluihin ja prosesseihin, joita käytetään web-sovelluskehityksen automaatiotestauksessa. Tutkimuksessa esitellään kohteena olleen sovelluksen kehitysputki, kuinka se muotoutuu ja toimii yleisiä web-sovelluskehitys ratkaisuja mukaillen.

Tämä tutkielma esittää vaihtoehtoisena ratkaisuna testiautomaation lisäämistä kehitysputkeen jo katselmointivaiheessa. Tämä mahdollistaa jatkuvan testiautomaation ylläpidon ja kehittämisen osaksi kehitystyötä, jossa testiautomaation huoltaminen sisältyy kehittäjien koodimuutosten ja regressiovaikutusten jatkuvaan seuraamiseen. Useamman ihmisen vastuulla olevat testit estävät testien kasvamisen liian suureksi yksittäiselle henkilölle, mikä jakaa vastuun useammalle henkilölle. Testauksen määrä kasvaa, ja mahdollisten virheiden havaitseminen tapahtuu varhaisessa vaiheessa kehitystä, mikä parantaa ohjelmiston teknistä laatua ja vähentää kuluja. Nopea palaute auttaa kehittäjiä oppimaan luomaan toimivampia muutoksia.

Haasteena testauksen määrän kasvattamisessa on testien vakauden ja muutosten etenemisnopeuden ylläpitäminen, kun testejä ajetaan jokaisen kehittäjän muutoksille. Katselmointiputkessa tulisi välttää ylimääräistä testiajojen odotusta tai saman asian tuplaverifiointia, mikäli muutos on jo todettu toimivaksi. Tulevaisuudessa testien ajamista tulisi voida ohjata helpommin haluttuihin muutoksiin. Testaus ja kehitysprosessit voivat vaihdella eri yritysten toimintatapojen ja ohjelmistojen luonteiden mukaan. Jokaisen ohjelmiston ja toimintaympäristön vaatimukset määrittelevä, kuinka laajasti ohjelmiston testaus tulee toteuttaa.

Tämä tutkielma esittelee katselmointiputken käytäntöjä vaihtoehtona testiautomaation kattavuuden ja vakauden parantamisessa, sekä vastuun jakamisessa. Tutkielmassa esitellyt testausprosessit ja työkalut ovat laajalti käytössä olevia vaihtoehtoja laadukkaalle testaamiselle.

8. LÄHTEET

[1] Thomson Reuters, 2022: 5 trends financial services organizations will face in 2022 — ready or not. https://legal.thomsonreuters.com/en/insights/articles/5-trends-financial-services-organizations-will-face-in-2022. Viitattu 29.9.2022.

[2] OP Ryhmä: Itseohjautuva toimintatapa. https://www.op.fi/op-ryhma/tietoa-ryhmasta/op-lyhyesti/kettera-toimintatapa. Viitattu 29.9.2022.

[3] Tom Hall, Devops Pipeline. Atlassian. https://www.atlassian.com/devops/devops-tools/devops-pipeline. Viitattu 29.9.2022.

[4] Vadapalli, S. (2018). DevOps: continuous delivery, integration, and deployment with DevOps: dive into the core DevOps strategies. Packt Publishing Ltd.

[5] Matt Heusser, 2021: 6 ways to harness test automation in a CI/CD pipeline. Techtarget. https://www.techtarget.com/searchsoftwarequality/tip/6-ways-to-harness-test-automation-in-a-CI-CD-pipeline. Viitattu 29.9.2022.

[6] Finanssivalvonta, 2008: Laki Finanssivalvonnasta. Finlex. https://www.finlex.fi/fi/laki/ajantasa/2008/20080878 Viitattu 29.9.2022.

[7] Arachchi, S. A. I. B. S., & Perera, I. (2018, May). Continuous integration and continuous delivery pipeline automation for agile software project management. In 2018 Moratuwa Engineering Research Conference (MERCon) (pp. 156-161). IEEE.

[8] Shanika Wickramasinghe, 2021: How To Set Up a Continuous Integration & Delivery (CI/CD) Pipeline. BMC blogi. https://www.bmc.com/blogs/ci-cd-pipeline-setup/. Viitattu 17.10.2022.

[9] Continuous integration. Wikipedia-artikkeli. https://en.wikipedia.org/wiki/Continuous_integration. Viitattu 29.9.2022.

[10] Debroy, V., Miller, S., & Brimble, L. (2018, October). Building lean continuous integration and delivery pipelines by applying devops principles: a case study at varidesk. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 851-856).

[11] Continuous delivery. Wikipedia-artikkeli. https://en.wikipedia.org/wiki/Continuous_delivery. Viitattu 29.9.2022.

[12] Donca, I. C., Stan, O. P., Misaros, M., Gota, D., & Miclea, L. (2022). Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects. Sensors, 22(12), 4637.

[13] Jenkins. Jenkins User Documentation. https://www.jenkins.io/doc/. Viitattu 12.10.2022.

[14] Django Stars (käyttäjänimi), 2017: Continuous Integration. CircleCI vs Travis CI vs Jenkins. Medium. https://medium.com/hackernoon/continuous-integration-circleci-vs-travis-ci-vs-jenkins-41a1c2bd95f5. Viitattu 18.10.2022.

[15] OpenJDK, 2022: Compilation Overview. https://openjdk.org/groups/compiler/doc/compilation-overview/index.html.Viitattu 1.1.2023.

[16] Apache Groovy. https://groovy-lang.org/. Viitattu 22.10.2022.

[17] Sandeep Kinayath, 2019: Tutorial: Jenkins Pipeline file with Apache Groovy. https://www.eficode.com/blog/jenkins-groovy-tutorial. Viitattu 22.10.2022

[18] Newman Liam, 2017: Converting Conditional Build Steps to Jenkins Pipeline. Blogi. https://www.jenkins.io/blog/2017/01/19/converting-conditional-to-pipeline/. Viitattu 19.10.2022.

[19] N. A. Sulaiman and M. Kassim, "Developing a customized software engineering testing for Shared Banking Services (SBS) System," 2011 IEEE International Conference on System Engineering and Technology, 2011, pp. 132-137, doi: 10.1109/ICSEngT.2011.5993436.

[20] A. Contan, C. Dehelean and L. Miclea, "Test automation pyramid from theory to practice," 2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2018, pp. 1-5, doi: 10.1109/AQTR.2018.8402699.

[21] Ham Vocke, 2018: The Practical Test Pyramid. Martin Fowler. https://martinfowler.com/articles/practical-test-pyramid.html. Viitattu 29.9.2022.

[22] M. Sánchez-Gordón, L. Rijal, and R. Colomo-Palacios. Beyond Technical Skills in Software Testing: Automated versus Manual Testing. 2020 In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20). Association for Computing Machinery, New York, NY, USA, 161–164. https://doi.org/10.1145/3387940.3392238

[23] Tarhini, A., Ismail, Z., & Mansour, N. (2008, December). Regression testing web applications. In 2008 International Conference on Advanced Computer Theory and Engineering (pp. 902-906). IEEE.

[24] Itkonen, J., Mantyla, M. V., & Lassenius, C. (2009, October). How do testers do it? An exploratory study on manual testing practices. In 2009 3rd International Symposium on Empirical Software Engineering and Measurement (pp. 494-497). IEEE.

[25] Bitbucket. Bitbucket Code Review. https://bitbucket.org/product/features/code-review/. Viitattu 29.9.2022.

[26] Kimmo Brunfeldt, 2021: A complete guide to code reviews. Swarmia. https://www.swarmia.com/blog/a-complete-guide-to-code-reviews/. Viitattu 29.9.2022.

[27] F. Fuxing, H. Daqing, Z. Wei, "Research and realize of automation test framework based on Web," Electronic Design Engineering, vol. 20, pp. 36-38, Oct. 2012.

[28] Sun, Z., Zhang, Y., & Yan, Y. (2019, December). A Web testing platform based on hybrid automated testing framework. In 2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC) (Vol. 1, pp. 689-692). IEEE.

[29] Parmar D, Exploratory testing, Atlassian. https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing. Viitattu 29.9.2022.

[30] Mobaraya, F., & Ali, S. (2019). Technical Analysis of Selenium and Cypress as Functional Automation Framework for Modern Web Application Testing. Department of Information Technology, AGI Institute, Auckland, New Zealand.

[31] Jaleel, H. Q. (2019). Testing Web Applications. SSRG International Journal of Computer Science and Engineering, 6(12), 1-9.

[32] Kaushik, D. M., & Fageria, P. (2014). Performance testing tools: A comparative study. International Journal of Innovative Science, Engineering & Technology.

[33] Stackify, 2021: The Ultimate Guide to Performance Testing and Software Testing: Testing Types, Performance Testing, Steps, Best Practices, and More. https://stackify.com/ultimate-guide-performance-testing-and-software-testing/. Viitattu 7.10.2022.

[34] Satyabrata Jena, 2021: Failover Testing in Software Testing. Geeks for geeks -sivusto. https://www.geeksforgeeks.org/failover-testing-in-software-testing/. Viitattu 7.10.2022.

[35] Li, J. (2020). Vulnerabilities mapping based on OWASP-SANS: a survey for static application security testing (SAST). arXiv preprint arXiv:2004.03216.

[36] EUR-Lex, 2016: Euroopan parlamentin ja neuvoston asetus. https://eur-lex.europa.eu/legal-content/FI/TXT/?qid=1528874672298&uri=CELEX%3A02016R0679-20160504. Viitattu 1.10.2022.

[37] Tuomo Yli-Huttula, 2022: Suomalaisten pankkien tietoturva on kansainvälisesti korkeatasoista. Finanssiala. https://www.finanssiala.fi/kolumni/suomalaisten-pankkien-tietoturva-on-kansainvalisesti-korkeatasoista/. Viitattu 1.10.2022.

[38] Kev Zettler. DevSecOps Tools. Atlassian. https://www.atlassian.com/devops/devops-tools/devsecops-tools. Viitattu 1.10.2022.

[39] Croft, R., Newlands, D., Chen, Z., & Babar, M. A. (2021, October). An empirical study of rule-based and learning-based approaches for static application security testing. In Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (pp. 1-12).

[40] Yang, E. (2018, April). Fuzz testing & software composition analysis in software engineering. In 2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT) (pp. 1-3). IEEE.

[41] Snyk. SAST vs. SCA testing: What's the difference? Can they be combined? https://snyk.io/learn/application-security/sast-vs-sca-testing/. Viitattu 1.10.2022.

[42] Yksikkötestaaminen. Wikipedia-artikkeli. https://fi.wikipedia.org/wiki/Yksikk%C3%B6testaaminen. Viitattu 2.10.2022.

[43] Khorikov, V. (2020). Unit Testing Principles, Practices, and Patterns. Simon and Schuster.

[44] Katalon. What is Regression Testing? Definition, Tools & How to Get Started. https://katalon.com/resources-center/blog/regression-testing. Viitattu 29.9.2022.

[45] M. Wahid and A. Almalaise, "JUnit framework: An interactive approach for basic unit testing learning in Software Engineering," 2011 3rd International Congress on Engineering Education (ICEED), 2011, pp. 159-164, doi: 10.1109/ICEED.2011.6235381.

[46] Dirk Riehle. 2008. JUnit 3.8 documented using collaborations. SIGSOFT Softw. Eng. Notes 33, 2, Article 5 (March 2008), 28 pages. https://doi.org/10.1145/1350802.1350812

[47] D. D. Ma'ayan, "The Quality of Junit Tests: An Empirical Study Report," 2018 IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE), 2018, pp. 33-36.

[48] Ravindranath Harita, 2021: Jest vs Mocha vs Jasmine: Comparin The Top 3 JavaScript Testing Framewoks. Lambdatest-artikkeli. https://www.lambdatest.com/blog/jest-vs-mocha-vs-jasmine/. Viitattu 7.10.2022.

[49] MochaJS. https://mochajs.org/. Viitattu 7.10.2022.

[50] Fabisiak Radek, 2022: Mocha vs Jest Comparison of Testing Tools in 2022. Duomly-blogi. https://www.blog.duomly.com/mocha-vs-jest/. Viitattu 7.10.2022.

[51] Meta, 2022: Jest. https://jestjs.io/. Viitattu 2.10.2022.

[52] Sunil Sandhu, 2019: I tested a React app with Jest, Enzyme, Testing Library and Cypress. here are the differences. https://javascript.plainenglish.io/i-tested-a-react-app-with-jest-testing-library-and-cypress-here-are-the-differences-3192eae03850. Viitattu 2.10.2022.

[53] Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., & Pinto, G. (2019, May). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC) (pp. 209-219). IEEE.

[54] JUnit 5. https://junit.org/junit5/. Viitattu 29.9.2022.

[55] SonarQube. SonarQube Documentation. https://docs.sonarqube.org/latest/. Viitattu 29.9.2022.

[56] Mitre Corporation, 2023: CVE-2022-42004. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-42004. Viitattu 1.1.2023.

[57] Jfrog. CI-CD Integration with Xray. https://www.jfrog.com/confluence/display/JFROG/CI-CD+Integration+with+Xray. Viitattu 29.9.2022.

[58] CyperRes, 2022: Fortify Static Code Analyzer (SCA) Static Application Security Testing. Microfocus.com. https://www.microfocus.com/media/data-sheet/fortify_static_code_analyzer_static_application_security_testing_ds.pdf. Viitattu 7.10.2022.

[59] Selenium. Selenium automates browsers. That's it! What you do with that power is entirely up to you. https://www.selenium.dev/. Viitattu 29.9.2022.

[60] Github. Jbehave. https://github.com/jbehave/jbehave-core. Viitattu 29.9.2022.

[61] Python. https://www.python.org/. Viitattu 29.9.2022.

[62] Robot framework. Introduction. Robot Framework dokumentaatio. https://robotframework.org/. Viitattu 29.9.2022.

[63] Jbehave. What is Jbehave. https://jbehave.org/. Viitattu 29.9.2022.

[64] Github. Robot Framework. https://github.com/robotframework/robotframework. Viitattu 29.9.2022.

[65] Cypress. The web has evolved. Finally, testing has too. https://www.cypress.io/. Viitattu 29.9.2022.

[66] Khan, R. B. (2016). Comparative study of performance testing tools: apache Jmeter and HP loadrunner.

[67] LoadRunner and Performance Center, 2018: System Requirements. https://admhelp.microfocus.com/lre/en/12.60-12.63/pdfs/PC_LR_System_Requirements_12.61.pdf. Viitattu 2.10.2022.

[68] Apache. Apache Jmeter. https://jmeter.apache.org/. Viitattu 2.10.2022.

[69] Blazemeter. Welcome to BlazeMeterDocs. https://guide.blazemeter.com/hc/en-us. Viitattu 2.10.2022.

[70] Locust.io. What is Locust. https://docs.locust.io/en/stable/. Viitattu 2.10.2022.

[71] Yuri Bushnev, 2020: Jmeter vs Locust. https://www.blazemeter.com/blog/jmeter-vs-locust. Viitattu 2.10.2022.

[72] Software Testing Help 2022: WebLOAD Review – Getting Started With WebLOAD Load Testing Tool. https://www.softwaretestinghelp.com/webload-load-testing-tool-review/. Viitattu 2.10.2022.

[73] Justina Alexandra Sava, 2022: Leading vendors' share in the IT infrastructure monitoring software market worldwide in 2022. Statista.com. https://www.statista.com/statistics/1258480/it-infrastructure-monitoring-software-market-share-vendor-worldwide/. Viitattu 2.10.2022.

[74] IBM, 2021: Dynatrace documentation. https://www.ibm.com/docs/en/mcmp/211201?topic=configuration-dynatrace. Viitattu 2.10.2022.

[75] The Apache Software Foundation, 2023: Apache Maven Project. https://maven.apache.org/. Viitattu 1.1.2023.

[76] StackOverflow. https://stackoverflow.com/questions/33319992/how-to-stop-robot-framework-test-execution-if-first-testcase-fail. Viitattu 22.10.2022.

[77] Testivetoinen kehitys. Wikipedia-artikkeli. https://fi.wikipedia.org/wiki/Testivetoinen_kehitys. Viitattu 27.12.2022.