Tampereen yliopisto

Antti Hautajärvi

CI/CD SULAUTETTUJEN JÄRJESTELMIEN OHJELMISTOKEHITYKSESSÄ

Kandidaatintutkielma
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja yliopistonlehtori Erja Sipilä
Toukokuu 2024

TIIVISTELMÄ

Antti Hautajärvi: CI/CD sulautettujen järjestelmien ohjelmistokehityksessä
Kandidaatintutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaatin tutkinto-ohjelma, sähkötekniikka
Toukokuu 2024

Sulautetut järjestelmät ovat tärkeä osa modernia yhteiskuntaa. Niiden käyttökohteita löytyy niin ajoneuvoista, lääketieteestä, viihde-elektroniikasta kuin tuotantolaitteista. Sulautettujen järjestelmien ohjelmakoodien koko ja monimutkaisuus kasvavat jatkuvasti, minkä takia tarvitaan keinoja tehokkaan ja luotettavan ohjelmistokehityksen takaamiseksi. Jatkuva integraatio (CI), toimitus (CDE), sekä käyttöönotto (CD) ovat metodeja, joita voidaan hyödyntää näihin tarpeisiin. Nämä edellä mainitut jatkuvat työtavat ovat menetelmiä, joilla pyritään automatisoimaan ohjelmistokehitystä ja tehostamaan koodin integrointia, testausta ja käyttöönottoa. Tämän kandidaatintutkielman tavoitteena on selvittää kirjallisuuden avulla jatkuvien työtapojen ja niiden pohjalta luotujen järjestelmien vaikutus sulautettujen järjestelmien ohjelmistokehitykseen.

Työ alkaa sulautettujen järjestelmien ja ohjelmistojen perusperiaatteiden sekä käyttökohteiden esittelyllä. Tämän aikana todetaan sulautettujen järjestelmien omaavan monia etuja ja rajoitteita perinteisiin tietokonejärjestelmiin verrattuna. Työssä esitellään Agile ja DevOps, jotka ovat jatkuvien työtapojen taustalla olevia työmalleja. Niiden pohjalta perehdytään tarkemmin CI:n, CDE:n ja CD:n hyödyntämiseen ohjelmistokehityksessä. Lisäksi työssä käsitellään kahta järjestelmää: Jenkins ja Gitlab CI/CD, jotka on luotu jatkuvan kehityksen mahdollistamiseksi. Tutkielmassa todetaan molempien järjestelmien omaavan etuja ja heikkouksia toisiinsa verrattuna.

Toteutetun kirjallisuuskatsauksen perusteella CI/CD-menetelmien käyttö sulautettujen järjestelmien kehityksessä voi olla hyödyllistä ja kannattavaa projektin koosta ja tarpeista riippuen. Jatkuvan testauksen ansiosta ohjelmistovirheiden määrä vähenee ja asiakastyytyväisyys kasvaa. Sulautetut järjestelmät toimivat usein toimintakriittisissä ympäristöissä, joten ohjelmistovirheitä ei saisi tapahtua. Lisäksi työssä todetaan, että CI/CD:n avulla valmista ohjelmakoodia saadaan asiakkaalle useammin. Tämän ansiosta kehittäjät saavat aiempaa tiheämmin palautetta työstään, jonka pohjalta pystytään kehittämään tulevia implementointeja.

CI/CD:n toteuttaminen ja ylläpitäminen ei ole kuitenkaan täysin ongelmatonta. Tutkielman perusteella CI/CD-järjestelmien integrointi olemassa oleviin järjestelmiin ja laitteisiin voi tuottaa haasteita. Lisäksi todetaan, että automatisoitujen testien luominen on työlästä ja työntekijöiden osaaminen voi olla puutteellista. Näiden ongelmakohtien ratkaiseminen vaatii esimerkiksi investointeja ja työkulttuurin muutosta. Tutkielman tulosten perusteella voidaan kuitenkin todeta, että jatkuvien työtapojen, kuten CI/CD:n käyttäminen, on perinteiseen ohjelmistokehitykseen verrattuna tehokkaampaa.

Avainsanat: Gitlab CI/CD, jatkuva integraatio, jatkuva käyttöönotto, jatkuva toimitus, Jenkins, sulautettu järjestelmä

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

CD	Continuous Deployment, jatkuva käyttöönotto
CDE	Continuous Delivery, jatkuva toimitus
CI	Continuous Integration, jatkuva integraatio
DevOps	Development & Operations
Git	Yleinen versionhallintajärjestelmä
HW	Hardware, laitteisto
SDLC	Software Development Life Cycle, ohjelmistokehityksen elinkaari
SW	Software, ohjelmisto
VCS	Version Control System, versionhallintajärjestelmä

1. JOHDANTO

Sulautetut järjestelmät muodostavat olennaisen osan nykyaikaista teknologiaa, löytyen laajasti eri sovellusalueilta kuten teollisuudesta, lääketieteestä, ja kulutuselektroniikasta. Näiden järjestelmien monimutkaisuus ja yksilöllisyys aiheuttavat haasteita niiden ohjelmistokehitykselle. Sulautettujen järjestelmien ohjelmakoodien koko kasvaa jatkuvaa tahtia, joten ohjelman automaattinen testaaminen, vahvistaminen ja julkaiseminen ovat tärkeässä asemassa. Jatkuva integraatio (engl. Continuous Integration, CI) ja jatkuva käyttöönotto (engl. Continuous Deployment, CD) ovat käytäntöjä, jotka tarjoavat ohjelmistokehitystiimeille ratkaisuja näihin haasteisiin.

CI/CD-liukuhihna (engl. pipeline) on automatisoitu prosessi, joka seuraa lähdekoodin siirtymistä versionhallintajärjestelmästä siihen asti, kunnes se on tuotannossa käyttäjien käytettävissä [1]. CI/CD menetelmien tarkoituksena on mahdollistaa jatkuva ja automatisoitu ohjelmistoprosessi, joka parantaa ohjelmistokehityksen tehokkuutta, tuotteen laadunhallintaa sekä mahdollistaa paremman asiakastyytyväisyyden.

Tämä kandidaatintyö tutkii jatkuvan integraation ja jatkuvan käyttöönoton hyödyntämistä sulautettujen järjestelmien ohjelmistokehityksessä. Työ toteutetaan kirjallisuus-katsauksena. Luvussa 2 esitellään sulautetut järjestelmät ja ohjelmistot sekä niiden sovelluksia. Luku 3 käsittelee ketteriä työmenetelmiä, kuten DevOpsia sekä CI/CD:tä ja esitellään niihin kuuluvia ominaisuuksia. Luvussa 4 vertaillaan kahta erilaista järjestelmää, jotka on luotu CI/CD- liukuhihnan toteuttamiseksi. Lisäksi luvussa 4 käsitellään testaamisen merkitystä jatkuvissa työtavoissa. Luvussa 5 käsitellään hyötyjä, joita ilmenee jatkuvan integraation ja -käyttöönoton hyödyntämisestä, sekä käydään läpi niiden implementointiin liittyviä haasteita. Luku 6 muodostaa yhteenvedon.

2. SULAUTETUT JÄRJESTELMÄT

Sulautetut järjestelmät ovat tärkeä osa nykyaikaista teknologiaa, ja niitä käytetään laajasti eri aloilla monimutkaisten toimintojen suorittamiseen. Usein sulautetut järjestelmät jäävät laitteen käyttäjältä huomaamatta, tai niiden toiminnallisuus ei ole esillä. Alaluvussa 2.1 käsitellään sulautettuja järjestelmiä yleisesti, niiden käyttötarkoituksia sekä haasteita kehitysprosessin aikana. Alaluvussa 2.2 käydään läpi sulautettujen ohjelmistojen kehittämistä ja niihin liittyviä ominaisuuksia.

2.1 Sulautetut järjestelmät yleisesti

Sulautettu järjestelmä koostuu yleisesti ohjelmistosta (engl. Software, SW) ja laitteistosta (engl. Hardware, HW). Ohjelmiston tehtävänä on ohjata laitteistoa reaaliaikaisesti halutulla tavalla. Yksinkertaistettuna ohjelmisto pitää sisällään silmukkarakenteen, joka ottaa vastaan käyttäjän antamia syötteitä ja toteuttaa näiden perusteella tarvittavia toimintoja. [2, s .17]. Tämä ohjelmisto sisältää käyttöjärjestelmän, sovelluksia, ajureita ja muita tarvittavia ohjelmistokomponentteja. Laitteisto pitää sisällään kaikki fyysiset komponentit, joita järjestelmään kuuluu. Tällaisia komponentteja ovat esimerkiksi prosessorit, muistit, napit ja anturit.

Sulautetut järjestelmät voivat olla osa laajempaa järjestelmää, kuten ajoneuvoa, lääketieteellistä laitetta tai älykotia. Tällöin järjestelmät toimivat yhtenä osana laajempaa kokonaisuutta ja ovat vastuussa tietyn toiminnallisuuden tai tehtävän suorittamisesta. Chattopadhyayn mukaan suurin osa sulautetuista järjestelmistä on luotu suorittamaan vain yhtä tehtävää toistuvasti [3, s. 2]. Esimerkiksi potilaan sydämen toimintaa mittaava sydänmonitori, joka antaa hälytyksen poikkeavuuksista, on tällainen järjestelmä.

Sulautetut järjestelmät voidaan yleisesti jakaa kahteen eri pääryhmään: riippumattomiin ja riippuvaisiin järjestelmiin. Riippuvat järjestelmät ovat sulautettuja järjestelmiä, jotka toimivat koordinoidusti tietokonejärjestelmien tai muiden sulautettujen järjestelmien kanssa [2, s. 11]. Muutamia esimerkkejä riippuvista sulautetuista järjestelmistä:

- Langattomat kuulokkeet, jotka hallitsevat äänen toistoa ja vastaanottoa langattomasti kuulokkeiden sekä puhelimen välisen Bluetooth-yhteyden avulla.
- GPS-navigaattori, joka vastaanottaa signaaleja satelliitilta ja näyttää käyttäjälle sijaintitietoja sekä reittiohjeita.

- Robottipölynimuri, joka on yhteydessä kodin Wifi-verkkoon, minkä kautta imurin toimintaa voidaan ohjata ja ajastaa.
- Älykäs termostaatti, joka säätää huoneen lämpötilaa käyttäjän asetusten ja ympäristön olosuhteiden mukaisesti.

Toisaalta sulautettu järjestelmä voi toimia itsenäisesti ilman ulkoisia lähteitä tai yhteyksiä. Tällöin se toimii sisäisten prosessien ja ohjelmoitujen toimintojen mukaisesti suorittaakseen määritellyt tehtävät. Esimerkiksi teollisuusrobotiikassa sulautetut järjestelmät voivat suorittaa ennalta määrättyjä tehtäviä täysin itsenäisesti ilman jatkuvaa ulkoista ohjausta tai tiedonvaihtoa. Tällainen autonomisuus on tärkeää alueilla ja paikoissa, joissa ei voida taata ulkoisien yhteyksien toimivuutta, kuten syrjäisillä alueilla ja avaruudessa. [2, s. 11]

Toisin kuin yleiset tietokonejärjestelmät, sulautetut järjestelmät ovat erikoistuneita järjestelmiä, jotka on suunniteltu suorittamaan tarkasti määriteltyjä tehtäviä suljetussa tai rajoitetussa ympäristössä. Tämän tarkan rajauksen ansiosta sulautettujen järjestelmien suunnittelu ja valmistaminen on halvempaa kuin yleisen tietokonejärjestelmän [2, s. 10]. Lisäksi sulautetut järjestelmät käyttävät toimiessaan vähemmän energiaa, joten ne soveltuvat hyvin käytettäviksi kannettavissa ja pienikokoissa laitteissa [2, s. 19].

Järjestelmän tarkka rajaus voi myös tuottaa vaikeuksia systeemin suunnittelulle. Rajoitettujen järjestelmien täytyy usein olla optimoituja esimerkiksi akunkeston, koon tai valmistushinnan suhteen [3, s. 2]. Nämä lisävaatimukset tuovat omia haasteita niin laitteiston kuin ohjelmiston suunnitteluun ja kehittämiseen. Esimerkiksi ohjelmiston osalta on tärkeää pyrkiä minimoimaan muistin ja suoritustehon käyttö samalla säilyttäen tarvittava toiminnallisuus [3, s. 3]. Tämä edellyttää tarkkaa harkintaa siitä, mitkä ominaisuudet sisällytetään järjestelmään ja miten ne toteutetaan mahdollisimman tehokkaasti.

Sulautetut järjestelmät ovat vuosi vuodelta monimutkaisempia, monipuolisempia, sekä niiden suorittamat tehtävät voivat olla toiminta- ja turvallisuuskriittisiä [4]. Täten asiakkaan tarpeiden kartoitus, järjestelmän valmistus, testaaminen ja julkaisu ovat erittäin tärkeitä vaiheita sulautetun järjestelmän kehityksessä.

2.2 Sulautettujen ohjelmistojen kehittäminen

Ohjelmisto on avainasemassa sulautetussa järjestelmässä, sillä se tuo laitteeseen älyä, funktionaalisuutta ja kontrollia suunnitellun käyttötarkoituksen toteuttamiseksi [2, s, 27]. Laitteisto toimii järjestelmän fyysisenä osana, jota ohjelmisto ohjaa

käyttäjän antamien syötteiden ja ohjelmoijan määrittämien toimintalogiikoiden perusteella. Sulautettujen järjestelmien ohjelmistojen luomiseen käytetään samoja ohjelmistokieliä kuin muidenkin ohjelmistojen kehittämiseen. Sulautetuissa ohjelmistoissa on kuitenkin paljon vaatimuksia ja rajoitteita, jotka tekevät tehtävästä vaativan ja eroavan perinteiseen ohjelmistokehitykseen verrattuna.

Jeongin et. al mukaan nykyaikaisen ohjelmistokehityksen metodit olettavat ohjelman toimivan yhdellä laitteella, ilman viiveongelmia tai resurssien kulumista [5, s. 1]. Sulautettujen järjestelmien ohjelmistokehityksessä tulee kuitenkin ottaa nämäkin asiat huomioon, joten kehityskulku on erityyppistä kuin perinteisessä ohjelmistokehityksessä. Näitä aiemmin mainittuja vaatimuksia kutsutaan ei-toiminnallisiksi vaatimuksiksi [5, s. 2]. Ei-toiminnalliset vaatimukset eroavat toiminnallisista siten, että ne eivät vaikuta suoranaisesti laitteen toimintaan tai tehtäviin. Nämä vaatimukset ovat silti kriittisiä laiteen suorituskyvyn, turvallisuuden, luotettavuuden ja käytettävyyden kannalta. Sulautettujen ohjelmistojen valmistamiselle on yleistä, että laitteistoalusta on määritetty ennen ohjelmiston valmistamista, koska merkittävä osa koodista riippuu siitä [5, s. 1]. Laitteistossa voi olla akunkestoon, suorituskykyyn, muistiin ja fyysiseen kokoonpanoon liittyviä rajoitteita, jotka pitää ottaa huomioon ohjelmistoa tehdessä.

Sulautettuja ohjelmistoja voidaan kehittää useilla eri ohjelmointikielillä. Oikean kielen valinta riippuu ohjelmiston tarpeista, laitteistosta sekä kehittäjien osaamisesta ja kokemuksesta. [2, s. 31] Yleisimmät ohjelmointikielet sulautettujen ohjelmistojen kehittämiselle ovat C ja C++. Ne tarjoavat tehokkaan alustan, jolla luoda suoraviivaista ja kompaktia ohjelmakoodia. Muita yleisiä ohjelmointikieliä sulautetuille ohjelmistoille ovat Java ja Python. Java on suosittu vaihtoehto, mikäli järjestelmä vaatii nettiyhteyttä tai mobiilisovellustukea. Python vuorostaan on saanut paljon suosiota sulautettujen ohjelmien kehittäjien keskuudessa sen yksinkertaisuuden ja joustavuutensa ansiosta. [2, s. 32] Lopullisen alustan ja ohjelmointikielen valinta on tärkeä päätös, ja se tulee tehdä harkitusti sekä onnistuneesti sulavan ohjelmistokehityksen ja tehokkaan sulautetun järjestelmän toiminnan varmistamiseksi.

3. KETTERÄT TYÖTAVAT JA CI/CD

Sulautettujen järjestelmien ohjelmistokehitys muodostaa monimutkaisen prosessin, joka vaatii erityisiä lähestymistapoja ja työkaluja. Sulautettu järjestelmä koostuu ohjelmistosta sekä laitteistosta ja ohjelmistojen kokojen kasvaessa uudenlaisia metodeja ohjelmakoodin testaamiseen ja julkaisemiseen vaaditaan. Ketterien työtapojen sekä CI/CD:n ominaisuudet voivat auttaa vähentämään monimutkaisuutta, lisäämään tehokkuutta sekä virtaviivaistamaan työnkulkua [6]. Tässä luvussa esitellään perusteet Agilesta, DevOpsista, jatkuvasta integraatiosta (CI) sekä jatkuvasta toimituksesta (CDE) että käyttöönotosta (CD).

3.1 Agile

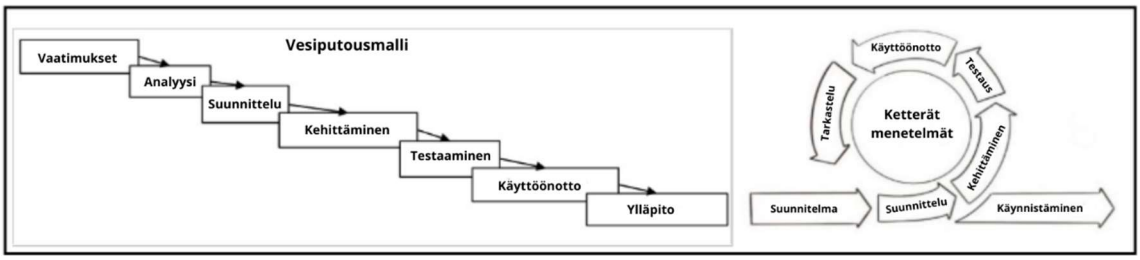
Ketterät työtavat, tunnettu englanninkielisellä nimityksellä Agile, ovat nykyaikainen lähestymistapa ohjelmistokehitykseen. Agilen tarkoituksena on ylittää vaikeudet ja haitat, jotka ovat tyypillisiä perinteisten menetelmien soveltamisessa ohjelmistohankkeiden hallintaan ja toteutukseen [7]. Ideana on luoda iteratiivinen ja inkrementaalinen ohjelmistokehitysprosessi, joka korostaa jatkuvaa vuorovaikutusta asiakkaiden kanssa, nopeaa reagointia muutoksiin ja tiivistä tiimityöskentelyä. Täten ohjelmistotiimit voivat reagoida nopeasti projektien muuttuviin vaatimuksiin ja asiakastarpeisiin, mikä johtaa parempaan lopputulokseen ja asiakastyytyväisyyteen.

Ketterien työtapojen alkuperänä pidetään vuonna 2001 julkaistua *ketterän ohjelmistokehityksen julistusta* (Agile Manifesto) [8]. Julistus sisältää arvoja ja periaatteita, joita tulisi noudattaa modernissa ja ketterässä ohjelmistokehityksessä. Ydinideana julistuksessa ovat seuraavat arvot [8]:

- **Yksilöitä ja kanssakäymistä** enemmän kuin menetelmiä ja työkaluja
- **Toimivaa ohjelmistoa** enemmän kuin kattavaa dokumentaatiota
- **Asiakasyhteistyötä** enemmän kuin sopimusneuvotteluja
- **Vastaamista muutokseen** enemmän kuin pitäytymistä suunnitelmassa

Agilen lisäksi yksi suosittu ja yleinen ohjelmistokehityksen metodi on vesiputousmalli. Tässä mallissa työvaiheet tehdään samanaikaisesti ja testaaminen suoritetaan yleensä prosessin lopussa, kuten on esitetty kuvassa 1. Tämä johtaa siihen, että kun jokainen seitsemästä vaiheesta on suoritettu, kehittäjät etenevät seuraavaan vaiheeseen. Kun uuteen vaiheeseen on edetty, aikaisempaan ei voida palata ainakaan hylkäämättä koko

projektia ja aloittamatta sitä alusta. [9, s. 777] Tämän takia vesiputousmalli ei ole houkutteleva työtapa nykyaikaisessa ohjelmistokehityksessä kasvavien kustannusten, ja hitaamman markkinoillesaannin vuoksi.



Kuva 1. *Ohjelmistokehityksen kaari: vesiputousmalli vs. Agile, muokattu lähteestä [9, s. 777]*

Sulautettujen ohjelmistojen kehittämisessä vesiputousmallilla voi olla hyötyjä erityisesti tilanteissa, joissa järjestelmän vaatimukset ovat vakaat ja selkeästi ennalta määritellyt. Tällaisissa tapauksissa vesiputousmalli mahdollistaa prosessin selkeän ja helposti ennustettavan etenemisen, mikä voi olla tärkeää tietyissä kehitysympäristöissä. Kuitenkin on huomattava, että Agile tarjoaa joustavamman ja nopeamman kehityksen, joka samalla sitoo asiakkaan projektiin tiukemmin.

Agile kattaa useita erilaisia työtapoja ja käsitteitä, kuten DevOps, jatkuva integraatio, jatkuva toimitus ja jatkuva käyttöönotto, joita käsitellään tarkemmin luvussa 3.2. Vaikka nämä käsitteet ovatkin olennainen osa modernia ohjelmistokehitystä, Agilea voidaan pitää laajempana filosofiana ja menetelmänä, johon nämä käytännöt perustuvat [9, s. 776].

3.2 DevOps

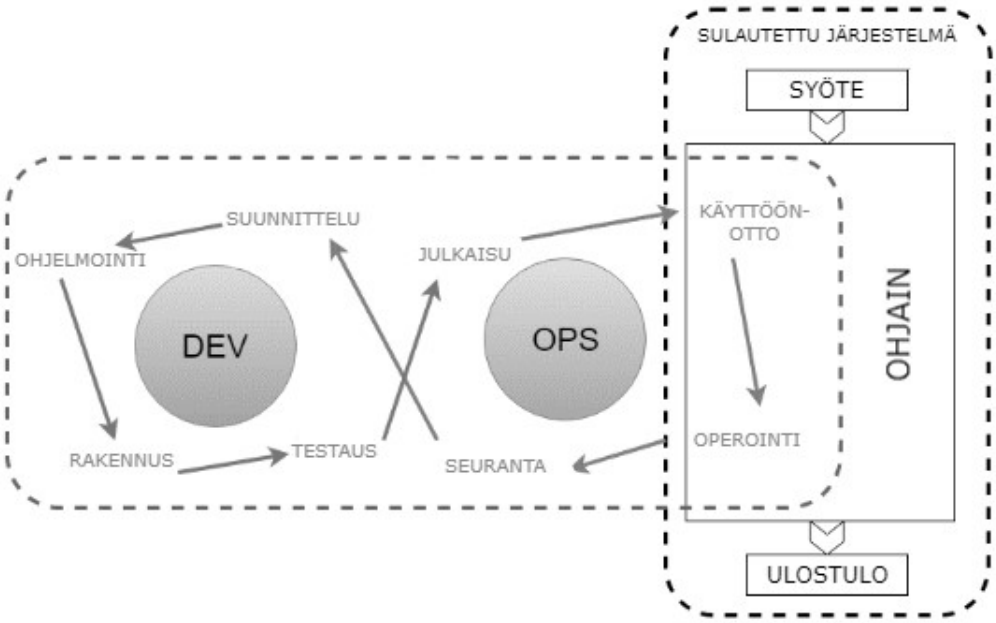
Ketterät työtavat ovat nostaneet suosiotaan merkittävästi ohjelmistokehityksessä viimeisen kahdenkymmenen vuoden aikana [10]. Yksi suosituimpia ja tärkeimpiä osia ketteristä menetelmistä on DevOps. Sen perimmäinen idea on tiivistää kehittäjien ja tuotannon välistä yhteistyötä. Lyhenne DevOps tulee englanninkielisistä sanoista Development & Operations, jotka tarkoittavat kehitystä ja tuotantoa. DevOps sisältää paljon erilaisia toisiinsa liittyviä toimintamalleja ja tapoja, joista Ghantous & Gill pitävät tutkimuksessaan tärkeimpinä ihmisten kommunikaatiota ja yhteistyötä, jatkuvaa toimitusta sekä automatisoitua liukuhihnaa [11, s. 8].

Teknologian kehittyessä jatkuvasti ja nopeasti, tulee myös sulautettujen järjestelmien ohjelmistojen olla nopeasti päivitettäviä ja korjattavia. Wijaya et. al mukaan klassista järjestelmäkehityksen elinkaarta (SDLC, engl. Software DEvelopment Life Cycle) ei

voida enää hyödyntää optimaalisesti nykyaikaisessa sulautettujen järjestelmien kehityksessä [12, s. 2]. SDLC:n sijasta nykyaikaisessa ohjelmistokehityksessä pyritään suosimaan DevOps:in hyödyntämistä. DevOps:in käyttö sulautettujen järjestelmien kehityksessä tuo mukanaan useita hyötyjä: [7][12]

- Mahdollistaa nopean julkaisu- ja käyttöönottosyklin.
- Parantaa kehittäjien, tuotannon ja yrityksen sidosryhmien välistä kommunikaatiota.
- Lyhentää päivitysten saamista valmiiseen tuotteeseen.
- Vähentää ohjelmistovirheiden määrää ja nopeuttaa virheiden löytämistä.
- Lisää tuottavuutta ja tiimin suorituskykyä.

Almeidan et. al teettämän tapaustutkimuksen mukaan automaatio, kommunikaatio ja markkinoilletuloaika ovat tärkeimpiä ketterien menetelmien ja DevOps:in hyötyjä [7]. Automaatio pitää sisällään CI/CD liukuhihnan, ja tapaustutkimuksen mukaan kahdeksan 12:sta organisaatiosta pitävät sitä tärkeänä osana nykyaikaista ohjelmistokehitystä [7].



Kuva 2. *DevOps:n käyttö sulautetuissa järjestelmissä, muokattu lähteestä [12, s. 3].*

Sulautettujen järjestelmien ja DevOps:n välinen yhteys ilmenee järjestelmän käyttöönoton ja käyttämisen aikana. DevOps-menetelmässä eri työvaiheet toteutuvat iteratiivisesti sykleissä, mikä tiivistää yhteistyötä kehitys- ja operatiivisten tiimien välillä. [12, s. 3] Tätä työtapaa on havainnollistettu kuvassa 2.

3.3 Ohjelmistokehityksen automaatio

Ohjelmistokehityksen automatisointi on olennainen osa nykyaikaista kehitysprosessia, joka pyrkii parantamaan tehokkuutta, laadunhallintaa ja ohjelmiston nopeaa toimittamista markkinoille. Tässä luvussa tarkastellaan kolmea käsitettä, jatkuvaa integraatiota (CI, engl. Continuous Integration), jatkuvaa toimitusta (CDE, engl. Continuous Delivery), sekä jatkuvaa käyttöönottoa (CD, engl. Continuous Deployment), jotka muodostavat perustan ohjelmistokehityksen automaatiolle. Ensimmäisessä alaluvussa käsitellään CI:n periaatteet ja toisessa keskitytään CDE:n ja CD:n rooleihin.

3.3.1 Jatkuva integraatio

CI/CD-liukuhihnan ensimmäinen vaihe on jatkuvan integraation toteuttaminen. CI on laajasti käytetty työtapa, jossa kehittäjät integroivat koodinsa jaettuun arkistoon (engl. repository) monesti päivän aikana [13, s. 3]. Täten ohjelmoijat saavat palautetta koodin toiminnallisuudesta nopeasti ja jatkuvasti, ja voivat tehdä tarvittavia muutoksia tehokkaasti. CI pitää sisällään ohjelmakoodin rakentamisen ja testaamisen, ja muut CI/CD liukuhihnan metodit voidaan sisällyttää CD:hen [13, s. 4].

Pittettin mukaan jatkuva integraatio painottaa suuresti testiautomaatiota, jolla tarkistetaan, ettei sovelluksen toiminta rikkoudu integroitaessa muutoksia päähaaraan [14]. Näin ennaltaehkäistään virheiden pääsyä lopulliseen tuotteeseen ja vähennetään niistä aiheutuvien ongelmien korjaamista. Jos virheellinen koodi pääsee kuitenkin päähaaraan asti, voidaan ongelma löytää helpommin muutoksen ollessa pieni. Lisäksi pienten muutosten sisältämien vikojen korjaaminen on nopeampaa verrattuna isoihin muutoksiin.

Jatkuva integraatio tarvitsee toimiakseen automatisoituja testejä jokaiselle uudelle ominaisuudelle, parannukselle ja korjaukselle [14]. Täten kehittäjien tulee tuntea testausympäristöt sekä testipenkkien toiminta. Testien tulee olla kattavia ja luotettavia, jotta ne voivat havaita mahdolliset virheet ajoissa ennen ohjelmakoodin pääsyä valmiiseen tuotteeseen. Pittettin mukaan ohjelmakoodin testaaminen CI-liukuhihnassa tulisi tapahtua aina puskettaessa muutoksia päähaaraan, tai vaihtoehtoisesti myös jo jokaisen ohjelmoijan omassa haarassa [15].

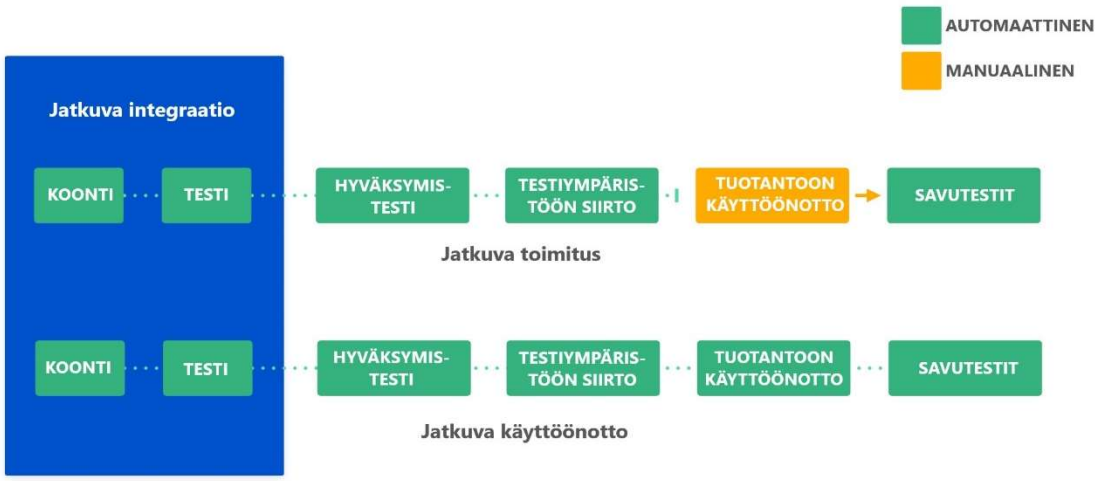
Ohjelmistokoodien muutoksissa ja kehittämisissä kuitenkin tapahtuu suurella todennäköisyydellä virheitä, niin pieniä kuin suurempiakin. McAllisterin mukaan nämä virheet voivat jäädä ihmiseltä, kuin koneeltakin huomaamatta ja niitä voi päästä tuotantoon asti. Ohjelmistovirheet ilmenevät monin eri tavoin, kuten kääntämisvirheinä,

funktionaalisuuksien rikkoutumisina tai käyttäjien hyväksynnän epäonnistumisina. McAllisterin mukaan vian korjaaminen tai aiemman ohjelmakoodin palauttaminen ovat ainoat tavat edetä näissä tilanteissa. [16] Kun jatkuva integraatio on toteutettu onnistuneesti, ja kehittäjät sekä tuotanto ovat yhteisymmärryksessä sen käytöstä, on ongelmien korjaaminen mahdollista tehdä entistä edullisemmin ja nopeammin.

3.3.2 Jatkuva toimitus ja käyttöönotto

Jatkuvan käyttöönoton pohjimmaisena tarkoituksena on saada valmista ohjelmakoodia asiakkaan käyttöön mahdollisimman usein ja nopeasti. Tavoitteena on saada uusimmat toiminnallisuudet asiakkaalle mahdollisimman varhaisessa vaiheessa ja samalla oppia heidän käyttötavoistaan. [17] Tämä mahdollistaa ohjelmiston kehittäjille ja organisaatiolle nopean reagoinnin käyttäjän tarpeisiin tämän antaman palautteen mukaisesti.

Lyhenne CI tarkoittaa jatkuvaa integraatiota, kun taas CD voidaan mieltää olevan kaksi eri asiaa, jatkuva toimitus tai jatkuva käyttöönotto. Näiden kahden käsitteen ero löytyy niiden toteutustavasta. Jatkuva integraatio on itsessään osa molempia CD:n toteutustapoja, mutta kehitysputken tuotantoonsaantivaiheessa on eroja tapojen välillä. Tämä eroavaisuus on esillä kuvassa 3. Joissain kirjoituksissa jatkuvasta toimituksesta voidaan käyttää myös lyhennettä CDE tai vastaavasti jatkuvasta käyttöönotosta lyhennettä CDT [5] [13] [18]. On myös huomioitavaa, että useassa tutkimuksessa ja julkaisussa lyhennettä CD on käytetty kuvaamaan molempia tapoja, eikä lyhenteet CDE/CDT ole vakiintunut kirjallisuudessa. Tässä tutkimuksessa käytetään jatkuvalle toimitukselle lyhennettä CDE ja jatkuvalle käyttöönotolle lyhennettä CD.



Kuva 3. *Jatkuvan toimituksen ja käyttöönoton vaiheet sekä erot, muokattu lähteestä [14].*

Jatkuvan toimituksen ja käyttöönoton avainero on siis ohjelman tuotantoon käyttöönotossa. CD ottaa automaation yhden askeleen pidemmälle liukuhihnassa verrattuna CDE:hen, täten mahdollistaen täysin automatisoidun ketjun. Shahin tarkentaa julkaisussaan, että CDE:ssä sovellus on mahdollisesti julkaisukelpoinen päivityksen jälkeen, kun taas CD:ssä sovellus on automaattisesti julkaistu jokaisen päivityksen mukana. CD-liukuhihna on automatisoitu prosessi, joka ei sisällä manuaalisia vaiheita tai ihmisen tekemiä päätöksiä. Näin ollen, kun kehittäjä on sitonut muutoksensa, tämä muutos siirtyy tuotantoympäristöön ilman ihmisen väliintuloa. [18, s. 1]

Täysin automatisoitu julkaisuprosessi vaatii kehittäjätiimiltä ja organisaatiolta tarkkaan määrättyjä sääntöjä, joiden perusteella ohjelmakoodia luodaan ja julkaistaan [18]. Koska uuden päivityksen julkaisua valvoo automatisoitu ohjelma eikä ihminen, täytyy kehittäjien luottaa koodin automaattiseen testaamiseen. Täten testien tulee olla kattavat, jotta virheitä ei pääse tuotantoon asti. Ohjelmakoodin testaaminen tapahtuu liukuhihnan CI-vaiheessa, kuten esillä kuvassa 3, joten CI:n tulee olla luotu onnistuneesti ennen CD:n implementointia.

Taulukko 1. *Yhteenveto häiriötekijöistä siirryttäessä CDE:stä CD:hen, muokattu lähteestä [18, s. 115].*

VAIKUTTAVA TEKIJÄ	#	%
Täysin automatisoidun hyväksymistestin puute	43	43,9
Manuaalinen laaduntarkastus	42	42,9
Käyttöönotto liiketoimintapäätöksenä	41	41,8
Riittämätön automaattisen testauksen kattavuus	34	34,7
Erittäin byrokraattinen käyttöönottoprosessi	31	31,6
Tehokkaan peruutusmekanismin puuttuminen	24	24,5
Riippuvuus sovellustasolla	23	23,5
Motivoitumaton asiakas	19	19,4
Asiakasympäristö	16	16,3
Toimialan rajoitteet	15	15,3
Testitulosten manuaalinen tulkinta	11	11,2

Liukuhihnan muuttaminen CDE:stä täysin automatisoituun ja toimivaan CD:hen voi olla hyvinkin vaikea prosessi. Taulukossa 1 on esillä Shahinin et al. tuottaman

tapaustutkimuksen tulokset yleisimmistä tekijöistä, jotka vaikeuttavat tai estävät CD:n toteuttamista. Kyselytutkimuksessa oli mukana 98 osallistujaa, sisältäen muun muassa arkkitehtejä, konsultteja, tiimin johtajia ja kehittäjiä. [18, s. 113] Taulukosta 1 selviää, että täysin automatisoidun hyväksymisprosessin puute on merkittävä este, joka vaikuttaa yli 40 prosenttiin tapauksista. Tämä korostaa automaation tärkeyttä prosessissa. Samalla manuaalinen laaduntarkastus sekä riittämätön automaattisen testauksen kattavuus ovat keskeisiä haasteita, mikä viittaa testausprosessin tehostamisen tarpeisiin. Käyttöönotto liiketoimintapäätöksenä ja erittäin byrokraattinen käyttöönottoprosessi viittaavat organisaatioiden sisäisiin haasteisiin, joiden osuudet ovat myös huomattavat.

Suurin houkute toteuttaa muutos CDE:stä CD:hen on CD:n kyky saada ohjelmistojen muutokset nopeammin käyttöön, nopea palautteen saaminen ja ongelmien korjaaminen edullisemmin verrattuna CDE:hen [19, s. 56]. Shahinin et al. tuottamasta tapaustutkimusta selviää, että useat yritykset ja tahot ovat skeptisiä tämän päivityksen kannattavuudesta. Automatisoidun käyttöönoton tuottaminen on monimutkainen prosessi, joka ei ole välttämättä aina kannattava. [18, s. 115] Liukuhihnan päivittäminen täysin automatisoituun CD-muotoon vaatii organisaatiolta paljon osaamista ja ymmärrystä siihen käytettävistä työkaluista. Näitä työkaluja käsitellään tarkemmin luvussa neljä.

4. CI/CD:N TOTEUTTAMINEN

Sulautettuja järjestelmiä tuottavilla yrityksillä on tapana seurata kaavaa jatkuvan integraation ja käyttöönoton toteuttamisessa. Aluksi yrityksissä voi herätä mielenkiinto ketteriin työtapoihin, joiden pohjalta työskentelyä aletaan toteuttamaan sprinteissä. Tästä seuraa jatkuvan integraation hyödyntäminen, jossa ohjelmistoa integroidaan ja testataan jatkuvaan tahtiin. CI-liukuhihna johtaa yleisesti siihen, että valmista ohjelmakoodia lähetetään tuotantoon alkuperäistä tiheämpään tahtiin. Täten jatkuva käyttöönotto vaikuttaa yrityksille loogiselta askeleelta kehityksessä. Näin asiakkaat saavat aiempaa nopeammin uusimman version ohjelmasta, ja samalla kehittäjät saavat arvokasta palautetta aiemmasta toteutuksesta. [17]

Ohjelmakoodien kokojen kasvaminen sulautetuissa ohjelmistoissa, sekä ohjelmisto-projekteissa yleisesti luo kasvavan tarpeen toteuttaa tehokas CI/CD liukuhihna, sekä hyödyntää DevOps:in oppeja. Onnistuneen CI/CD-liukuhihnan toteuttaminen ei kuitenkaan ole yksinkertainen prosessi. Ennen DevOps:in ja sen työmallien käyttöön-ottoa tulee organisaation ymmärtää tarkasti siihen liittyvät konseptit, työtavat, työkalut sekä hyödyt, että haasteet [20, s. 6]. Tässä luvussa keskitytään keskeisen DevOps:in työmallin, eli CI/CD:n toteuttamiseen valmistettuihin työkaluihin, Jenkinsiin sekä Gitlab CI/CD:hen.

4.1 Järjestelmät CI/CD:n toteuttamiseen

Markkinoilla on tarjolla useita työkaluja ja järjestelmiä tehokkaan CI/CD-liukuhihnan toteuttamiselle. Automatisoidut testit ovat olennainen osa CI/CD-liukuhihnaa, sillä ne varmistavat koodin toimivuuden ja laadun jokaisen muutoksen jälkeen. Erdenebatin mukaan DevOps:n lisätoteutuksissa, optimoinnissa ja parannuksissa tulisi priorisoida etenkin automaatiota [20]. Näin vältytään isojen virheiden pääsyltä tuotantoon ja vähennetään korjauksiin kuluvaa aikaa ja resursseja.

Ohjelmistojen kehittäjät luovat muutoksia ohjelmaan omassa versionhallintajärjestelmän (VCS, engl. Version Control System) haarassaan, josta valmista ohjelmakoodia voidaan integroida kehittäjien yhteiseen arkistoon. Tämän jälkeen muokattu koodiarkisto pitää rakentaa uudelleen, jotta saadaan projektin vaatimusten mukaiset suoritettavat tiedostot. Onnistuneen rakentamisen jälkeen koodia pitää testata kaikilla mahdollisilla testi-tapauksilla, jotta voidaan varmistaa sen toiminnallisuus. Ohjelmakoodin rakentaminen ja

testaaminen vievät hyvin paljon aikaa ohjelmistoprojekteissa ja lisäksi testitapauksien toistuva epäonnistuminen pidentää tätä aikaa ennestään. [21]

Jatkuvan integraation avuksi on luotu monia työkaluja, kuten Jenkins, Gitlab CI sekä Travis-CI, mutta nämä eivät itsessään edusta jatkuvaa integraatiota. On huomioitavaa, että jatkuva integraatio on käsitteenä laajempi, kuin pelkästään se työkalu, jolla liukuhihna luodaan. Sen sijaan CI tarkoittaa koodimuutosten jatkuvaa integrointia tiimin jäsenten kesken, millä vältytään laajamittaisilta yhteensulauttamiskonflikteilta (engl. Merge conflict) sekä kommunikaation puutteesta johtuvista virheistä. [16] Nämä järjestelmät ovat vain työkaluja tiimien käytettäväksi, jotta jatkuva kehityskulku saadaan toimimaan.

Seuraavissa alaluvuissa käsitellään kahta yleisintä sulautettujen järjestelmien ohjelmistokehityksessä käytettyä järjestelmää, jotka ovat luotu jatkuvan integraation sekä käyttöönoton toteuttamiseksi. Alaluku 4.1.1 käsittelee Jenkins:iä ja alaluku 4.1.2 Gitlab CI/CD:tä.

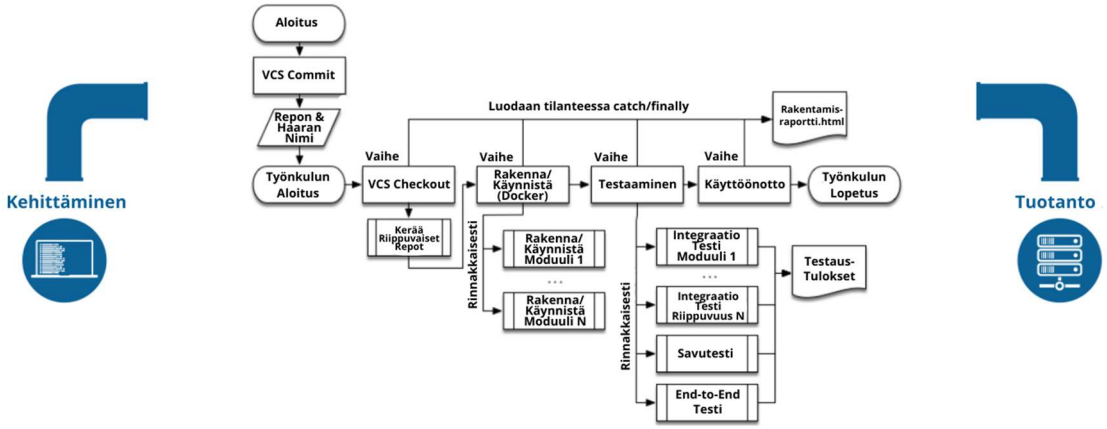
4.1.1 Jenkins

Jenkins on avoimen lähdekoodin CI/CD-työkalu, joka on rakennettu Java ympäristöön. Jenkins tukee projektien rakentamista, testaamista ja julkaisua virtuaalisesti. Tämä tekee siitä tehokkaan työkalun jatkuvan integraation ja käyttöönoton toteuttamiselle. Perus-periaatteena on automatisoida ohjelmiston kehitysprosessi mahdollisimman pitkälle CI/CD-menetelmien avulla. [16]

Jatkuvan integraation lisäksi Jenkinsiä voidaan hyödyntää jatkuvan käyttöönoton toteuttamiseen. Kun automatisoitua jatkuvaa käyttöönottoa otetaan käyttöön, tulee käyttöönottoautomaation vaiheet olla tarkasti standardoituja mahdollisimman pitkälle. Tavoitteena on luoda tiukasti noudatettavia julkaisu- ja käyttöönottoprosesseja, joiden pohjalta automatisaatio valmistetaan. Tämän avulla Jenkins voi valmistaa julkaisunäkökohtia ja suorittaa käyttöönottoja. [16]

Jenkins tarjoaa täysin automatisoituja ratkaisuja CI/CD-liukuhihnan luomiseksi. Kuvassa 4 on yksi esimerkki CI/CD-tilanteesta, joka on mallinnettu Jenkinsin avulla. Liukuhihnan alkupäässä on kehitysosasto, josta uutta ohjelmakoodia pyritään syöttämään tuotantoon liukuhihnan vaiheiden läpi. Jotta liukuhihna saadaan käyttöön, tulee ensin luoda versionhallintajärjestelmään sitoumus (engl. Commit). Git on tässä yleisesti käytetty versionhallintajärjestelmä. Tämän jälkeen VCS-repositorion ja haaran (engl. Branch) tiedot tulee syöttää Jenkinsiin, jotta liukuhihna voidaan konfiguroida oikein ja käyttöönotto aloittaa. [22]

VCS-checkout-vaiheessa Jenkins hakee VCS:stä ohjelmakoodin ja muut tarvittavat tiedostot. Tämä vaihe varmistaa, että Jenkins käyttää aina uusinta versiota ohjelmakoodista. Seuraavaksi suoritetaan ohjelmiston koostaminen ja/tai paketointi, tässä esimerkissä ne julkaistaan Docker-kontteihin. Seuraava vaihe on ohjelmakoodin testaaminen eri tavoilla. Erilaisia testausmetodeja ovat esimerkiksi savutestit ja integraatiotestit, kuten esillä kuvassa 4. Nämä testit suoritetaan rinnakkaisesti prosessin nopeuttamiseksi. Testeistä luodaan tiedosto, joka sisältää testien tulokset, kuten virheet, testien keston ja resurssien käytön. [22] Näiden tulosten avulla ohjelmakoodia voidaan joko korjata, tai parantaa seuraavassa iteraatiossa.



Kuva 4. *Esimerkki Jenkins-liukuhihnasta, muokattu lähteestä [22].*

Käyttöönottovaiheessa ohjelmisto julkaistaan joko manuaalisesti tai automaattisesti käyttäjän asetusten perusteella. Päivitetty ohjelmakoodi voidaan julkaista suoraan tuotantoon, tai VCS:n päälinjaan muiden kehittäjien saataville. Tämä riippuu siis siitä, onko Jenkins alustettu toimimaan CD:n vai CDE:n mukaisesti. Jotta ohjelma pääsee käyttöönottovaiheeseen, on sen tullut läpäistä kaikki testit toiminnallisuuden varmistamiseksi. [21] Kun käyttöönotto on suoritettu, ohjelma on joko valmis tai jo automaattisesti julkaistu tuotantoon. Kuten kuvassa 4 on esillä, jokaisen vaiheen aikana luodaan html-tiedosto, johon on sisällytetty dataa liukuhihnan kulusta ja ohjelman toiminnasta. Tiedostoon tulevaa tietoa voidaan rajoittaa käyttäjän asetusten perusteella ja tätä dataa voidaan hyödyntää jatkokehitystä varten. [22]

Jenkins suorittaa ennalta määrättyjä tehtäviä triggerien perusteella. Kun tietty tapahtuma tapahtuu tai aikaikkuna saavutetaan, triggeri laukaisee Jenkins-liukuhihnaan liittyvän työn (engl. Job). Triggeri voi olla esimerkiksi jokin ohjelmakoodiin tehty muutos versionhallintajärjestelmässä, kuten Git:ssä. Toisaalta triggeri voi olla myös aikataulutettu, joka laukaisee Jenkins-työn tiettyinä kellonaikoina tai päivinä. [21] Näiden

triggerien avulla Jenkins mahdollistaa automatisoidun ja jatkuvan työnkulkuprosessin, mikä säästää aikaa ja minimoi inhimillisten virheiden riskiä.

Gitlab CI/CD:hen verrattuna Jenkins on vaikeampi ja hitaampi alustaa. Jenkinsin tehokas käyttö vaatii lisäosia, kuten gitin ja Dockerin, kun taas Gitlab:ssa näitä alustamisia ei tarvita valmiin Git-ympäristön vuoksi. Tämä raskas alustamistyö kuitenkin johtaa siihen, että Jenkinsillä uuden sovellusversion käyttöönottaminen on nopeampaa, kuin Gitlab:n tarjoamalla jaetulla suoritusympäristöllä. Singhin et al. tuottaman tutkimuksen mukaan Jenkins on lähes 20-kertaisesti nopeampi suorittamaan ohjelmakoodin julkistamisen verrattuna Gitlab:n jaettuun suoritusympäristöön. Toisaalta Gitlab tarjoaa myös määritellyn suoritusympäristön, jonka avulla ohjelmakoodin julkaisu on yhtä nopeaa kuin Jenkinsillä. Määritellyn suoritusympäristön luominen on kuitenkin työläämpää ja sen käyttö voi viedä enemmän resursseja. [23, s. 11–12]

4.1.2 Gitlab CI/CD

Gitlab on suosittu avoimen lähdekoodin projektienhallintatyökalu, ja sen CI/CD- työkalut tarjoavat kattavan ja joustavan ratkaisun automatisoituun ohjelmistokehitykseen. Gitlab CI/CD:n ollessa integroitu osa jo valmiiksi olevaa git-versionhallintatyökalua, takaa se saumattoman ohjelmistokehitysprosessin automatisoinnin versionhallinnasta tuotantoon. Tämä parantaa merkittävästi kehitystiimin tehokkuutta ja tuotteiden laadunhallintaa. [24, s. 2] Toisaalta Gitlab-ympäristön riippuvuus voi tuottaa ongelmia, mikäli organisaatiolla on käytössään joku toinen VCS kuin git.

Gitlab CI/CD:n liukuhihna käynnistyy, kun Gitlab-repositorioon luodaan muutos. Itsessään Gitlab-liukuhihna on YML-tiedosto, joka sisältää sarjan suoritettavia töitä tai muita vaiheita, joita halutaan tehdä. Työt ovat samantyylisiä, kuin mitä Jenkinsin käytössä tapahtuu. Ne siis sisältävät useita eri vaiheita, kuten koodin kääntämisen, testaamisen ja sovelluksen paketoinnin. Vaiheet suoritetaan yksi kerrallaan ja seuraavaan vaiheeseen siirrytään, jos edellinen on päättynyt onnistuneesti. Esimerkiksi vasta testitapausten onnistuttua Gitlab CI/CD:n liukuhihnassa siirrytään Docker-tiedoston luomiseen, jota käytetään ympäristöriippuvaisten ongelmien välttämiseksi. [23, s. 10]

Gitlab CI/CD sisältää kolme erilaista määrittelytyyppiä suoritusympäristölleen (engl. Runner). Nämä kolme vaihtoehtoa ovat jaettu-, määritelty- ja ryhmäsuorittaja. Jaettu suorittaja palvelee kaikkia projekteja, määritelty palvelee vain tiettyä. Ryhmäsuorittaja on ikään kuin jaettu suorittaja, mutta se on jaettu tietyn ryhmän projekteille. Gitlab CI/CD:n tarjoama oletussuorittaja on muodoltaan jaettu. [23, s. 11] Suorittajan valinta vaikuttaa

CI/CD-liukuhihnan nopeuteen ja tehokkuuteen. Kaikki suorittajavaihtoehdot ovat ilmaisia käyttää, mutta Gitlab CI/CD:n käytön hinta syntyy valitusta suunnitelmasta. Ilmainen taso tarjoaa rajoitettuja ominaisuuksia, kun taas ”Premium” ja ”Ultimate” tarjoavat tehokkaamman ja räätälöidyn liukuhihnan sekä ylläpitämisen [25].

Toisin kun Jenkins, Gitlab CI/CD ei tue asennettavia lisäosia, ja koko konfigurointi tehdään itse putkistossa. Tämä tekee Gitlab CI/CD:stä hieman vähemmän monipuolisen verrattuna Jenkinsiin [23, s. 10]. Vaikka Jenkins tarjoaa laajan valikoiman lisäosia, Gitlab CI/CD:n yksinkertainen ja integroitu luonne voi olla tehokas vaihtoehto sulautettujen järjestelmien kehittämisessä. Tämä korostuu etenkin silloin, kun tarvitaan sujuvaa integraatiota versionhallintaan ja muuhun Gitlabin tarjoamaan toiminnallisuuteen. Gitlab CI/CD:n sekä Jenkinsin käytöstä sulautettujen järjestelmien kehityksessä ei ole tutkittu kovinkaan paljoa. Jotta kyseisten järjestelmien hyödyistä saisi tarkemman kuvan, tulisi suorittaa jatkotutkimuksia.

4.2 Testaus ja laadunvarmistus

Ohjelmakoodin testaaminen ja laadunvarmistus ovat tärkeässä asemassa jatkuvissa työtavoissa. Niiden rooli on varmistaa, että ohjelmistokehitysprosessi on tehokas, ja tuottaa luotettavia ja laadukkaita lopputuotteita. Jatkuvan integraation sisältämä jatkuva automaattinen testaaminen pyrkii varmistamaan ohjelmakoodin laadun. Tässä luvussa käsitellään erityyppisiä testejä, joita käytetään CI/CD-liukuhihnan yhteydessä. Lisäksi tässä luvussa selvitetään testaamisen hyödyt laadunvarmistukselle.

Yksi tärkeimmistä testaustyypeistä sulautettujen järjestelmien ympäristössä on yksikkötestaus. Arefeen & Schillerin mukaan yksikkötestaus on vähimmäisvaatimus testejä suorittaessa [24, s. 1]. Yksikkötestaamisen ideana on testata yksittäisiä ohjelmiston osia, kuten funktioita, metodeja tai luokkia erikseen. Tavoitteena on varmistaa, että kukin osa toimii odotetusti ja tuottaa oikean tuloksen kaikissa tilanteissa. Yksikkötestien tärkeys korostuu Arefeen & Schillerin tuottaman tutkimuksen mukaan jatkuvassa integraatiossa. Mikäli testit eivät löydä ohjelmakoodissa piileviä ongelmia, ne menevät automaattisen liukuhihnan avulla yhteiseen repositorioon asti [24, s. 4]. Tämä saattaa johtaa lukuisiin uusiin ongelmiin tuotannossa, sekä usea työtunti menee hukkaan virheen selvittämisessä ja korjaamisessa.

Muita sulautetun järjestelmän testaamisen vaiheita voivat olla muun muassa toiminnallinen testaus, suorituskykytestaus, käyttöliittymätestaus ja hyväksyntätestaus. Toiminnallinen testaus varmistaa, että ohjelmisto suorittaa odotetut toiminnot oikein. Testit voivat sisältää monenlaisia simulaatioita, jotka kuvaavat käyttäjän toimintaa.

Suorituskykytestaus on vuorostaan tärkeää, koska monet sulautetut järjestelmät ovat rajoitettuja tehonkulutuksen tai akunkeston suhteen. Käyttöliittymätestaus selvittää käyttöliittymän käyttäjäystävällisyyden ja toimivuuden. Hyväksyntätestaus toteutetaan yleensä viimeisenä ennen tuotantoon siirtymistä. Siinä varmistetaan asiakkaan kanssa, että ohjelmisto vastaa asiakkaan tarpeita ja odotuksia. [26, s. 3922] [27, s. 6]

Onnistuneen testaamisen mittaamiselle on monia eri lähestymiskohtia. Yksi tällainen tapa on selvittää tuotantoon asti päässeiden ohjelmistovirheiden määrä. Dileepkumar & Mathew selvittivät Jenkinsillä luodun CI/CD-liukuhihnan vaikutusta ohjelmakoodin laatuun [28]. Perehdytään tähän tutkimukseen tarkemmin. Taulukkoon 2 on kirjattu heidän tuottaman tutkimuksen sisältämät vertailut ohjelmistotuotannon kehityksestä, kun CI/CD-liukuhihna on otettu käyttöön Jenkinsin avulla.

Taulukko 2. *CI/CD:n sisältämän testaamisen ja laadunvarmistuksen hyödyt, muokattu lähteestä [28, s. 3–4].*

Mittari	**Ennen CI/CD:tä (kpl)**	**CI/CD:n jälkeen (kpl)**	**Kehitys (%)**
Tuotannossa havaitut viat (kuukausittain)	25	10	60
Keskimääräinen aika vikojen korjaamiseen (päiviä)	5	2	60
Ensimmäisellä kerralla hyväksyttyjen osuus (%)	70	95	36
Yhteensulauttamiskonfliktit (kuukausittain)	30	10	67
Konfliktien ratkaisemiseen käytetty aika (tunteja)	90	30	67

Taulukosta 2 voidaan päätellä, että CI/CD:n sisältämien automaattitestien vaikutus kehitysprosessin tehokkuuteen ja kestoon on suuri. Dileepkumarin & Mathewn mukaan nämä laadun parannukset johtuvat ensisijaisesti automaattisista testaus- ja validointiprosesseista [28, s. 4]. Tässä tutkimuksessa nämä oli toteutettu Jenkins-pohjaisen CI/CD-liukuhihnan avulla.

Testaaminen on tärkeä osa jatkuvia työtapoja. Tarvittavat testit tulee toteuttaa tarkasti ja kattavasti, jotta voidaan varmistaa ohjelman toiminnallisuus kaikissa mahdollisissa

tilanteissa. Täten voidaan varmistaa sulautetun järjestelmän laadukkuus. Automatisoitujen testien luomiseen, sekä ohjelmakoodin testaamiseen tulee varata tarpeeksi aikaa, jotta virheet havaitaan ennen järjestelmän pääsyä tuotantoon asti. Koska sulautetut järjestelmät suorittavat usein kriittisiä tehtäviä, ne eivät saa sisältää virheitä. Esimerkiksi sairaalaympäristössä olevien sulautettujen järjestelmien tulee olla testattu kokonaisvaltaisesti, jotta ne eivät käyttäydy ei-halutulla tavalla elintärkeässä ympäristössä.

5. HYÖDYT JA HAASTEET

Sulautettujen järjestelmien ohjelmistokehityksessä jatkuva integraatio ja jatkuva käyttöönotto ovat merkittäviä käytäntöjä, jotka tarjoavat useita etuja. Näitä metodeja käytettäessä oikeaoppisesti kehittäjät sekä organisaatiot voivat tehokkaasti hallita ohjelmistokehitysprosessejaan samalla varmistaen ohjelman laadun ja nopean kehityksen. Näiden työtapojen ja uusien järjestelmien integroiminen olemassa oleviin projekteihin ja käytäntöihin tuo mukanaan myös haasteita.

Kuva 5. *Erottelu CDE:n hyötyjen ja haasteiden välillä organisaatiossa, muokattu lähteestä [19, s. 57].*

Laukkanen et al. ovat erotelleet tutkimuksessaan CDE:n käyttöönottoon liittyviä hyötyjä ja haasteita eri vaiheiden mukaisesti, nämä esitetty kuvassa 5. Vaikkakin kyseinen tutkimus keskittyy jatkuvan toimituksen hyötyihin ja haasteisiin, voi kuvan 5 mukaista erottelua hyödyntää myös pohtimaan koko CI/CD-liukuhihnan kannattavuutta.

Tässä luvussa käsitellään CI/CD:n tuomia hyötyjä sekä sen käyttöönotossa ilmeneviä haasteita. Alaluku 5.1 keskittyy CI/CD:n tarjoamiin hyötyihin ja alaluku 5.2 haasteisiin, joita sen hyödyntämisessä voi tulla vastaan.

5.1 Hyödyt

Jatkuva integraatio sekä jatkuva käyttöönotto tarjoavat potentiaalisesti monia parannuksia ja hyötyjä perinteisiin ohjelmistokehitysmalleihin verrattuna. CI/CD:n tarjoamat hyödyt tulevat yleensä esille vasta työtapojen ymmärtämisen ja onnistuneen liukuhihnan luomisen jälkeen [19, s. 57]. Tämä muutos työtavoissa ja järjestelmissä voi olla yrityksille suuri harppaus, etenkin jos organisaatiossa ei hyödynnetä muita Agilen ja

DevOps:n oppeja entuudestaan. Tässä luvussa käsitellään CI:n lisäksi CDE:n ja CD:n hyötyjä sulautettujen järjestelmien ohjelmistokehityksessä.

Kuten kuvassa 5 on esillä, nämä hyödyt voidaan jakaa kahteen eri vaiheeseen riippuen organisaation liukuhihnan tilasta. Kun organisaatio ei ole vielä ottanut käyttöön jatkuvaa toimitusta, sillä on ongelmia, joita CDE voisi ratkaista. Nämä ongelmat eivät ole suoranaisesti CDE:hen liittyviä, mutta jatkuvan toimituksen tarkoituksena olisi selvittää nämä. Tällaisia ongelmia voisi olla esimerkiksi ympäristöjen yhteensopivuudet ja sovellusten riippuvuuksien hallinta. CDE:n tuomat hyödyt realisoituvat yleensä vasta kun organisaatio on jo ottanut sen käyttöön onnistuneesti. [19, s. 57] Tämä pätee muihinkin CI/CD-liukuhihnan tuomiin etuihin. Yrityksen on mahdollista pyrkiä ennustamaan etuja omassa toimialassaan, mutta lopulliset kustannukset ja niistä selviävät hyödyt tulevat voimaan vasta tässä muutoksessa onnistuttua.

Dakkak et al. tutkivat julkaisussaan jatkuvan käyttöönoton tuomia etuja ohjelmistointensiivisten sulautettujen järjestelmien kehitykselle [17]. Perehdytään tähän tutkimukseen tarkemmin. Havaintojen mukaan CD:n tuoma nopeus on yksi isoimpia etuja kehityksessä. Nopeus näkyy lyhyempänä käyttöönottoaikana, palautteen saamisena, vikojen tunnistamisena ja uusien ratkaisujen toimittamisena. [17, s. 937] Tässä hyödyt tulevat ikään kuin kehässä, seuraten toinen toistaan. Kun uutta ohjelmakoodia saadaan käyttäjälle käytettäväksi nopeammin, saadaan myös käyttäjän antamaa palautetta nopeammin ja usein. Näin käyttäjä voi antaa palautetta mahdollisista vioista, ja viat pystytään korjaamaan nopeammin. Uuden korjauksen saaminen tuotteeseen on myös täten nopeampaa. Tätä voidaan pitää uuden ratkaisun toimittamisena. Kun päivitysten tekeminen ja ohjelman julkaiseminen vie entistä vähemmän aikaa, jää aikaa myös kokeiluihin, ominaisuuksien arviointiin ja tyypillisiin ylläpitotoimiin [17, s. 938].

Toinen Dakkakin et al. tuottamassa tutkimuksessa esille nouseva hyöty on tuotettujen järjestelmien laatu. Heidän tekemän tapaustutkimuksen mukaan CD:n implementoiminen vähensi tuotantoon asti päässeiden ohjelmistovikojen määrää, sekä mahdollisti vikojen korjaamisen aiempaa edullisemmin. Tapaustutkimuksessa organisaation otettua CD:n onnistuneesti käyttöön väheni käyttöönoton jälkeen löydettyjen ohjelmistovirheiden määrä 63 %:lla. Lisäksi tarvittavien lisätutkimusten määrä käyttöönoton jälkeen väheni 27 %:lla. [17, s. 939–940] Kyseessä on siis huomattavan suuri parannus.

Mikäli organisaatio ei aio toteuttaa CD- tai CDE-liukuhihnaa, voi se kuitenkin hyötyä suuresti pelkän jatkuvan integraation tuomien parannuksien avulla. Kun automaattiset

testit toteutetaan jokaisen koodimuutoksen yhteydessä, virheiden tunnistaminen ja korjaaminen nopeutuvat. Tämä parantaa ohjelmiston laatua ja vähentää virheiden korjauskustannuksia myöhemmin kehitysprosessin aikana. Lisäksi CI mahdollistaa tiiviin yhteistyön kehittäjien välillä. Koodin jatkuva integrointi tarkoittaa sitä, että kehittäjät työskentelevät aina ajantasaisen version parissa, mikä vähentää konfliktien ja yhteensopimattomuuksien riskiä. [13] Kun testaaminen automatisoidaan, vähentyy manuaalisen työn tarve. Täten kehittäjillä on aikaa keskittyä esimerkiksi jonkin muun ominaisuuden parantamiseen, järjestelmien ylläpitämiseen tai asiakastarpeiden kartoittamiseen.

Kokonaisuudessaan CI, CDE ja CD tarjoavat tehokkaan ja luotettavan tavan kehittää sulautettuja järjestelmiä. Nämä työtavat parantavat ohjelman laatua, nopeuttavat kehitysprosessia, parantavat yhteistyötä ja lisäävät kehitystiimien tuottavuutta.

5.2 Haasteet

Jatkuvien työtapojen hyötyjen lisäksi niiden aiheuttamat haasteet voidaan jakaa kahteen ryhmään kuvan 5 mukaisesti. Organisaatiolla voi siis olla haasteita, jotka hidastavat tai täysin estävät CI/CD:n käyttöönottoa, sekä haasteita, jotka ilmenevät vasta käytön aikana. Shahinin et al. tuottivat tutkimuksen, missä käsitellään niin CI:n, CDE:n kuin CD:n käyttöönottoon ja ylläpitämiseen liittyviä haasteita [26]. Perehdytään tähän tutkimukseen tarkemmin.

Haasteita, joita voi herätä ennen käyttöönottoon tapahtumista on monia. Shahinin et al. tuottaman tutkimuksen mukaan tällaisia haasteita ovat muun muassa tarvittavien työkalujen, teknologioiden sekä asiantuntemuksen puute, yleinen muutoksen vastainen ilmapiiri ja skeptinen ajattelutapa jatkuviin työtapoihin. [26, s. 3925–3926] Työkalujen ja teknologioiden puute on mahdollista ratkaista investoinneilla, mikäli organisaatio kokee ne kannattaviksi. Sama pätee työntekijöiden kouluttamiseen tai vastaavasti uusien palkkaamiseen, joilla löytyy aikaisempaa osaamista CI/CD järjestelmistä ja työtavoista. Yleisen ajattelutavan muutos jatkuvia työtapoja kohtaan voi olla vaikeaa muuttaa ilman konkreettisia todisteita niiden hyödyistä.

CI/CD-liukuhihnan käytössä syntyviä haasteita voi myös olla useita. Kuten kuvassa 5 on eroteltuna, nämä haasteet ilmenevät siinä vaiheessa, kun organisaatio on jo ottanut liukuhihnan käyttöön. Laukkasen et al. tuottaman tutkimuksen mukaan tällaisia ongelmia voivat olla esimerkiksi teknisen velan lisääntyminen, alhaisempi luotettavuus testeissä ja aikapaineet [19, s. 58]. Tutkimuksessa käytiin läpi 35 eri tapausta, joista selvinneet ongelmat jaettiin seitsemään eri ryhmään, mistä yleisimpiä olivat testaaminen, integrointi

ja järjestelmäongelmat. Testausongelmia ilmeni 19:ssa tapauksessa (54,3 %), integraatio-ongelmia 16:ssa (45,7 %) ja järjestelmäongelmia 12:ssa tapauksessa (34,3 %). [19, s. 62–63] Voidaan siis todeta, että erinäiset ongelmat CD:n aikana ovat yleisiä, ja niitä voi ilmetä monessa eri muodossa.

Jatkuva integraatio painottaa vahvasti testaamisen automatisointia, ja Shahinin et al. toteaa tutkimuksessaan tämän olevan haastava toteuttaa. Koska testejä on paljon erilaisia, niiden automatisointi on hyvin aikaa vievää sekä työlästä. Lisäksi laitteiston ja ohjelmiston väliset riippuvuudet voivat tulla haasteeksi, sekä toteutetut testit voivat olla huonolaatuisia ja epäonnistua useasti. Tämä ei pelkästään hidasta liukuhihnan toimintaa, vaan myös voi laskea kehittäjien motivaatiota ja luottamusta automaattiseen jatkuvaan käyttöönottoon. [26, s. 3926–3927] Mårtensson et al. tuottaman tapaustutkimuksen mukaan kaikkia testejä ei välttämättä pysty täysin automatisoimaan [27]. Tutkimuksessaan he käsittelivät muun muassa ruotsalaisen Gripen-hävittäjän sulautettujen järjestelmien testaamista. He päätyivät siihen tulokseen, että tiettyjä testejä voidaan automatisoida, mutta esimerkiksi lentäjän omaa harkintaa monimutkaisten käyttöskenaarioiden arvioinnissa on erittäin vaikea korvata automaattisilla testeillä. [27, s. 5–6]

CI/CD-työtapoihin liittyy useita haasteita, jotka sulautettuja järjestelmiä valmistavan organisaation on voitettava luodakseen kannattavan kehityksen. Organisaation on tunnistettava nämä haasteet omassa ympäristössään ja toteutettava tarvittavat toimenpiteet niiden selvittämiseksi, jotta CI/CD-työtapojen tarjoamat hyödyt voidaan täysimääräisesti hyödyntää. Tämä voi sisältää investointeja järjestelmiin ja työntekijöiden osaamiseen, resurssien optimointia sekä prosessien parantamista.

6. YHTEENVETO

Jatkuva integraatio ja jatkuva käyttöönotto ovat vakiintuneet käytännöt modernissa ohjelmistokehityksessä, ja niiden soveltaminen sulautettujen järjestelmien kehityksessä tarjoaa sekä mahdollisuuksia että haasteita. Tämän kandidaatintyön tarkoituksena oli tutkia CI/CD-työtapoja ja niiden vaikutusta sulautettujen järjestelmien ohjelmisto-kehitykselle. Samalla käsiteltiin kahta eri järjestelmää, Jenkins ja Gitlab CI/CD, jotka on luotu CI/CD-liukuhihnan toteuttamista varten. Työ toteutettiin kirjallisuuskatsauksena.

Työssä todettiin, että jatkuvan kehityksen avulla pyritään automatisoimaan kehitysprosessia, parantamaan ohjelmiston laatua ja nopeuttamaan ohjelmakoodin toimitusta. CI:n perusajatuksena on ohjelmakoodin jatkuva integroiminen kehittäjien päähaaraan, mikä mahdollistaa nopeamman palautteen saamisen ja virheiden tunnistamisen aikaisessa vaiheessa. CD puolestaan laajentaa tätä ajatusta edelleen, mahdollistaen ohjelmiston automaattisen käyttöönoton. Tämän ansiosta valmis ohjelmisto päätyy asiakkaalle entistä nopeammin.

Jenkins ja Gitlab CI/CD todettiin olevan hyvin samantyylisiä työkaluja, molempien omatessa tietyt vahvuudet ja heikkoudet. Gitlab CI/CD:n käyttöönotto on Jenkinsiä helpompaa, se takaa saumattoman yhteistyön versionhallintatyökalun kanssa sekä siinä on helppokäyttöinen käyttöliittymä. Jenkins puolestaan on Gitlab CI/CD:tä monipuolisempi mahdollisten lisäosien ansiosta, se takaa nopeamman julkaisunopeuden ja lisäksi Jenkins on hyvin joustava työkalu. Suurempien järjestelmien kehityksessä Gitlab CI/CD on kuitenkin parempi työkalu. Oikean järjestelmän valinta kehitysympäristöksi riippuu organisaation tarpeista, osaamisesta sekä resursseista. On tärkeää arvioida mikä työkalu sopii parhaiten organisaation käyttöön ja tarpeisiin ennen päätöksen tekemistä. Tässä työssä käsiteltiin kahta yleisinä työkalua sulautettujen järjestelmien kehityksessä, mutta muitakin vaihtoehtoja on olemassa.

Työn toteutuksen aikana havaittiin, että CI/CD-työtapojen ja -järjestelmien etuja ei ole tutkittu laajasti sulautettujen järjestelmien kehityksessä. Jotta näiden menetelmien todelliset hyödyt ja haasteet saataisiin selville, tulisi toteuttaa lisää tutkimuksia, jotka perustuvat käytännön projekteihin. On huomattava, että vaikka näiden työtapojen käyttöönotto voi olla aluksi haasteellista, se on todennäköisesti kannattavaa pitkällä aikavälillä. Lisätutkimukset voivat tarjota syvempää ymmärrystä siihen, miten CI/CD voi parantaa sulautettujen järjestelmien kehitystä ja tuoda konkreettista hyötyä organisaatioille.

LÄHTEET

[1] I. Pereira, T. Carneiro, E. Figueiredo (2023). Exploring the Ci/Cd Pipeline in Floss Repositories of Embedded Iot Systems. Julkaisija: SSRN. Saatavilla (24.03.2024): https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4529908

[2] M. Ibraheem, S. Adrees (2023). Embedded Systems: For Engineers & Students. Julkaisija: Amazon Digital Services LLC – Kdp. Saatavilla (25.03.2024): https://www.google.fi/books/edition/Embedded_Systems/qa-WEAAAQBAJ?hl=fi&gbpv=0

[3] S. Chattopadhyay (2023). Embedded System Design, 3. painos. Julkaisija: PHI Learning Pvt. Ltd. Saatavilla (25.03.2024): https://books.google.fi/books?id=JuKwEAAAQBAJ

[4] A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, P. Marwedel (2014). Embedded systems development: from functional models to implementations, 1. painos. Julkaisija: Springer

[5] E. Jeong, D. Jeong, S. Ha (2021). Dataflow Model–based Software Synthesis Framework for Parallel and Distributed Embedded Systems. ACM Transactions on Design Automation of Electronic Systems, 26(5), 1–38. Saatavilla (10.04.2024): https://doi.org/10.1145/3447680

[6] What is CI/CD? (2023). Julkaisija: RedHat. Saatavilla (29.03.2024): https://www.redhat.com/en/topics/devops/what-is-ci-cd

[7] F. Almeida, J. Simões, S. Lopes (2022). Exploring the benefits of combining devops and agile. Future Internet, Vol.14(2), No. 63. Julkaisija: MDPI. Saatavilla (30.03.2024): https://www.mdpi.com/1999-5903/14/2/63

[8] K. Beck et al. (2001). Manifesto for Agile Software Development. Saatavilla (10.04.2024): https://agilemanifesto.org/

[9] S. Najihi, S. Elhadi, R. Abdelouahid, A. Marzak (2022). Software testing from an agile and traditional view, Procedia Computer Science, Vol. 203, pp. 775–782, ISSN 1877-0509. Saatavilla: (02.04.2024): https://www.sciencedirect.com/science/article/pii/S1877050922007219

[10] H. Edison, X. Wang, K. Conboy (2022). Comparing Methods for Large-Scale Agile Software Development: A Systematic Literature Review. IEEE Transactions on Software Engineering, Vol. 48, No. 8, pp. 2709–2731, 1 Aug. 2022

[11] G. B. Ghantous, A. Gill (2017). DevOps: Concepts, Practices, Tools, Benefits and Challenges. PACIS 2017 Proceedings. no. 96. Julkaisija: AIS. Saatavilla (07.04.2024): https://aisel.aisnet.org/pacis2017/96

[12] P.E. Wijaya, I. Rosyadi, A. Taryana (2019). An attempt to adopt DevOps on embedded system development: empirical evidence. Journal of Physics: Conference Series (Vol. 1367, No. 1, p. 012078). Julkaisija: IOP Publishing.

Saatavilla (07.04.2024): https://iopscience.iop.org/article/10.1088/1742-6596/1367/1/012078

[13] I.-C. Donca, O. P. Stan, M. Misaros, D. Gota, L. Miclea (2022). Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects. Sensors Vol.22(12), 4637-. Julkaisija: MDPI. Saatavilla (05.04.2024): https://www.mdpi.com/1424-8220/22/12/4637

[14] S. Pittet. Continuous integration vs. delivery. vs deployment. Julkaisija: Atlassian. Saatavilla (25.03.2024): https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

[15] S. Pittet. How to setup continuous integration. Julkaisija: Atlassian. Saatavilla (30.03.2024): https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration

[16] J. McAllister (2015). Mastering Jenkins. Julkaisija: Packt Publishing.

[17] A. Dakkak, D. I. Mattos, J. Bosch (2021). Perceived benefits of Continuous Deployment in Software-Intensive Embedded Systems. IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 2021, pp. 934–941. Saatavilla (10.04.2024): https://ieeexplore.ieee.org/abstract/document/9529712

[18] M. Shahin, M. A. Babar, M. Zahedi, L. Zhu (2017). Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, Canada, 2017, pp. 111–120

[19] E. Laukkanen, J. Itkonen, C. Lassenius (2017). Problems, causes and solutions when adopting continuous delivery—A systematic literature review. Information and Software Technology, Vol. 82, pp. 55–79. Saatavilla (27.03.2024): https://www.sciencedirect.com/science/article/pii/S0950584916302324

[20] B. Erdenebat, B. Bud, T. Batsuren, T. Kozsik (2023). Multi-Project Multi-Environment Approach—An Enhancement to Existing DevOps and Continuous Integration and Continuous Deployment Tools. Computers (Basel), Vol. 12(12), No. 254. Julkaisija: MDPI. Saatavilla (07.04.2024): https://doi.org/10.3390/computers12120254

[21] N. Seth, R. Khare (2015). ACI (automated Continuous Integration) using Jenkins: Key for successful embedded Software development. IEEE 2nd International Conference on Recent Advances in Engineering & Computational Sciences (RAECS), Chandigarh, India, 2015, pp. 1–6. Saatavilla (20.04.2024): https://ieeexplore.ieee.org/abstract/document/7453279/citations#citations

[22] User Handbook for Jenkins. Saatavilla (17.04.2024): https://www.jenkins.io/doc/

[23] C. Singh, N. Gaba, M. Kaur, B. Kaur (2019). Comparison of Different CI/CD Tools Integrated with Cloud Platform. IEEE 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2019, pp. 7–12. Saatavilla (18.04.2024): https://ieeexplore.ieee.org/document/8776985

[24] M. Arefeen, M. Schiller (2019). Continuous Integration Using Gitlab. URNCST, Vol. 3, 1–11. Saatavilla (17.04.2024): https://urncst.com/index.php/urncst/article/view/152

[25] Gitlab pricing. Saatavilla (22.04.2024): https://about.gitlab.com/pricing/

[26] M. Shahin, M. Babar, L. Zhu (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access, vol. 5, pp. 3909-3943. Saatavilla (22.04.2024): https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7884954

[27] T. Mårtensson, D. Ståhl, J. Bosch (2016). Continuous Integration on Applied to Software-Intensive Embedded Systems – Problems and Experiences. International Conference on Product-Focused Software Process Improvement. Saatavilla (23.04.2024): https://www.researchgate.net/publication/309706302_Continuous_Integration_Applied_to_Software-Intensive_Embedded_Systems_-_Problems_and_Experiences

[28] S. Dileepkumar, J. Mathew (2023). Transforming Software Development: Achieving Rapid Delivery, Quality, and Efficiency with Jenkins-Based CI/CD Pipelines. IEEE 2023 Annual International Conference on Emerging Research Areas: International Conference on Intelligent Systems (AICERA/ICIS), Kanjirapally, India, 2023, pp. 1–6. Saatavilla (23.04.2024): https://ieeexplore.ieee.org/document/10420251%20