TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

LAURI KORISEVA

JATKUVAN TOIMITUKSEN KÄYTTÖÖNOTTO OHJELMISTOPROJEKTISSA

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen

Tarkastaja ja aihe hyväksytty 1. lokakuuta 2018

TIIVISTELMÄ

LAURI KORISEVA: Jatkuvan toimituksen käyttöönotto ohjelmistoprojektissa
Tampereen teknillinen yliopisto
Diplomityö, 50 sivua, 2 liitesivua
Marraskuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Prof. Hannu-Matti Järvinen

Avainsanat: jatkuva toimitus, Jenkins, testausautomaatio

Ohjelmistokehittäjillä on yleinen tarve julkaista kehittämäänsä ohjelmakoodia nopeasti ja turvallisesti loppukäyttäjien saataville. Monet ohjelmistoalan yritykset julkaisevat ohjelmistotuotteita ja -palveluita perinteisen toimitusprosessin mukaisesti, mutta se sisältää lukuisia käytännöllisiä ongelmia. Jatkuva toimitus ratkaisee kyseiset ongelmat automatisoimalla julkaisua edeltävät toimenpiteet.

Tämän työn tavoitteena oli tutkia erilaisia jatkuvan toimituksen työkaluja ja toteuttaa valitulla työkalulla automatisoitu liukuhihna yrityksen ohjelmistoprojektille. Jatkuvaa toimitusta varten pystytettiin virtuaalikoneympäristö, johon toteutettiin sekä käännös- että demoympäristö. Valittua työkalua käyttämällä toteutettiin liukuhihna, joka suorittaa ohjelmakoodin päivittämisen, kääntämisen, tuottamisen sekä hyväksymistestauksen.

Toteutettu liukuhihna suorittaa määritellyt vaiheet säännöllisesti ongelmitta, ja se tarjoaa hyväksymistesteistä saatavia tuloksia. Liukuhihnaa on alettu hyödyntämään aktiivisesti kohdeprojektin kääntämisessä ja testauksessa. Liukuhihnaa on tarkoitus hyödyntää tulevissa julkaisuissa, mutta sitä ennen hyväksymistestausta täytyy kehittää ja tuotteesta pitäisi jakaa vastuuta tasapuolisemmin koko projektitiimin kesken.

ABSTRACT

LAURI KORISEVA: Applying continuous delivery to a software project
Tampere University of Technology
Master of Science Thesis, 50 pages, 2 Appendix pages
November 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Hannu-Matti Järvinen

Keywords: continuous delivery, Jenkins, test automation

Releasing code quickly and safely to the end users is a common problem faced by software developers. Many information technology companies apply the traditional delivery process in releasing software products and services despite the process containing multiple practical problems. Continuous delivery solves such problems by automating tasks preceding the release.

The goal of this thesis was to examine different tools for continuous delivery and to implement an automated pipeline in a company's software project by utilizing the chosen tool. A virtual machine environment containing environments for building and staging was built for the continuous delivery. By utilizing the chosen tool, a pipeline was implemented that updates, builds, deploys and applies acceptance testing to the code.

The implemented pipeline performs the specified steps flawlessly and provides results from the acceptance tests. The pipeline is now actively used in building and testing the target project. The pipeline will be utilized in the upcoming releases of the application. Nevertheless, the acceptance testing needs to be improved, and the responsibility for the product should be shared more evenly among the project team.

ALKUSANAT

Kiitän Atostek Oy:ta, joka tarjosi tälle työlle aiheen ja rahoituksen. Sen lisäksi kiitän Atostek Oy:ta sen tarjoamasta runsaasta ajasta työn toteuttamiselle ja kirjoittamiselle.

Kiitän sekä työni tarkastajaa Hannu-Matti Järvistä että ohjaajaa Matti Helmistä ohjaamisesta ja tukemisesta työn saralla. Kiitän myös Atostek Oy:n ylläpitotiimiä ja automaattitestien tekijöitä auttavaisuudesta ympäristön pystyttämisessä. Lisäksi kiitän muita työtovereitani heidän luomastaan kannustavasta ja rennosta työilmapiiristä.

Tampereella, 19.11.2018

Lauri Koriseva

SISÄLLYSLUETTELO

KUVALUETTELO

OHJELMALUETTELO

LYHENTEET JA MERKINNÄT

ASP.NET	Avoimeen lähdekoodiin perustuva viitekehys web-sovelluksille
ATDD	Acceptance test–driven development, suom. hyväksymistestivetoinen ohjelmistokehitys
C#	Microsoftin kehittämä oliopohjainen ohjelmointikieli
CD	Continuous Delivery (suom. jatkuva toimitus), jatkuviin julkaisuihin perustuva ohjelmistokehitysmenetelmä
CI	Continuous Integration (suom. jatkuva integraatio), jatkuviin integraatioihin perustuva ohjelmistokehitysmenetelmä
DevOps	Development and Operations, suom. kehitys ja toimenpiteet
eRA	e-ReseptiArkisto, Atostekin kehittämä pilvipalvelu, joka tarjoaa rajapinnan Kelan Kanta-palveluihin
GUI	Graphical User Interface, suom. graafinen käyttöliittymä
HTTP	Hypertext Transfer Protocol, tiedonsiirtoprotokolla
HTTPS	Hypertext Transfer Protocol Secure, HTTP:n laajennos, jossa kommunikaatio salataan
IP	Internet Protocol, Internet-kerroksen protokolla
Java	Tulkattava oliopohjainen ohjelmointikieli
IDE	Integrated Development Environment, suom. integroitu ohjelmointiympäristö
IIS	Internet Information Services, Microsoftin kehittämä palvelinohjelmistokokonaisuus web-sovelluksille
Jenkins	Avoimeen lähdekoodiin perustuva automatisointityökalu
MVC	Model-view-controller, mallin, näkymän ja käsittelijän eriyttämiseen perustuva ohjelmistoarkkitehtuurimalli
Python	Korkean tason tulkattava ohjelmointikieli
QA	Quality Assurance, suom. laadunvarmistus
RDP	Remote Desktop Protocol (suom. etätyöpöytäyhteys), Windows-käyttöjärjestelmien välinen yhteyskäytäntö
SaaS	Software as a Service, suom. ohjelmiston hankkiminen palveluna
SVN	Subversion, suosittu versionhallintajärjestelmä
SQL	Structured Query Language
TCP	Transmission Control Protocol, tietoliikenneprotokolla
VM	Virtual Machine (suom. virtuaalikone), tietokonetta emuloiva ohjelma
XML	Extensible Markup Language, rakenteellinen kuvauskieli

1. JOHDANTO

Ohjelmakoodin julkaiseminen nopeasti ja turvallisesti on yleinen ongelma, jonka ohjelmistokehittäjät kohtaavat ohjelmistoprojekteissa. Useimmissa projekteissa sovelletaan perinteistä toimitusprosessia, jossa ohjelmiston toimittaminen jaetaan kehittämis-, laadunvarmistus- sekä julkaisuvaiheiseen. Perinteinen toimitusprosessi sisältää kuitenkin lukuisia heikkouksia ja sen soveltaminen johtaa usein tyytymättömyyten niin kehittäjien kuin asiakkaidenkin osalta. [32]

Jatkuva toimitus ratkaisee perinteisessä toimitusprosessissa esiintyvät ongelmat tarjoamalla automatisointia. Ketterien menetelmien tutkija Jez Humble on esittänyt tarkan määritelmän jatkuvalle toimitukselle: "Jatkuva toimitus on kyky viedä kaikentyyppisiä muutoksia, kuten uusia ominaisuuksia, konfiguraatiomuutoksia, ohjelmointivirheiden (bug) korjaamisia ja kokeiluja, tuotantoon tai suoraan loppukäyttäjlle turvallisesti, nopeasti sekä kestävästi [32]". Jatkuvan toimituksen avulla ohjelmistotuotteista voidaan tuottaa laadukkaampia hyödyntäen säännöllisiä ja tehokkaita julkaisuja.

Työn tavoitteena on ottaa jatkuva toimitus käyttöön ohjelmistoprojektissa, jossa on toistaiseksi noudatettu perinteistä toimitusprosessia. Tavoitteen saavuttamiseksi työssä perehdytään tarkemmin ketteriin menetelmiin, työkaluihin sekä eri tapoihin pystyttää ympäristö toimitukselle. Sen jälkeen projektille valitaan parhaiten soveltuva jatkuvan toimituksen työkalu ja projektille pystytetään erillinen ympäristö, jossa jatkuva toimitus suoritetaan toteutettavan liukuhihnan avulla ja suorituksesta saadaan palautetta testaustulosten muodossa. Täten tulevat julkaisut voidaan suorittaa liukuhihnan tarjoaman palautteen perusteella.

Luvussa 2 tarkastellaan perinteistä toimitusprosessia ja kartoitetaan sen heikkoudet jatkuvaan toimitukseen verrattuna. Jatkuvan toimituksen käsitettä avataan yksityiskohtaisemmin ja sitä vertaillaan muihin samankaltaisiin menetelmiin. Lisäksi luvussa kerrotaan liukuhihnan toiminnasta, ympäristöjen eriyttämisestä ja niiden tehtävistä sekä eri testausmenetelmien soveltamisesta.

Luvussa 3 esitetään potentiaalisia työkaluratkaisuja, joita voidaan hyödyntää jatkuvassa toimituksessa. Tarkastelun kohteena on vain muutama suosittu vaihtoehto, koska jatkuvan toimituksen työkaluja on lähes rajattomasti saatavilla.

Luvussa 4 esitetään eRA-palvelu, jonka ympärille työssä rakennetaan jatkuvan toimituksen ympäristö. eRA on Atostek Oy:n kehittämä web-palvelu, jota käytetään pääasiassa sosiaali- ja terveydenhuollon toimintayksiköiden liittymiseen Kelan tarjoamiin Kanta-palveluihin. Luvussa valitaan perustellusti sopivat työkalut ja menetelmät jatkuvan toimituksen toteuttamiselle. Ympäristö suunnitellaan ja toteutetaan palvelulle soveltuvaksi.

Lisäksi jatkuvan toimituksen suorittavan liukuhihnan rakenne kuvataan yksityiskohtaisesti.

Luvussa 5 esitetään tulokset, jotka saatiin, kun liukuhihnalla olevat tehtävät suoritettiin. Hyväksymistestauksesta saatavat tulokset arvioidaan tuotteen julkaisun kannalta. Lisäksi luvussa arvioidaan jatkokehitystarpeita sekä jatkuvalle toimitukselle että yksittäisten tehtävien automatisoinnille yrityksen ohjelmistoprojekteissa.

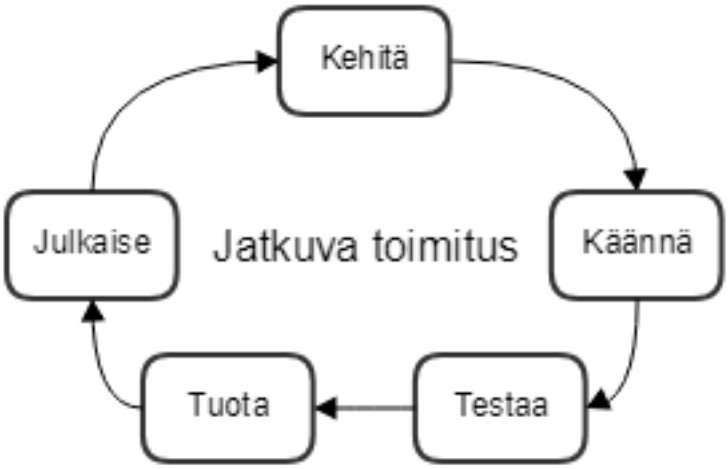
2. JATKUVA TOIMITUS

Jatkuva toimitus (Continuous Delivery, CD) on perinteisen määritelmän mukaan ohjelmistotuotannon menetelmä. Se on käsitteenä muilta osin hieman hämmentävä ja sen merkitys on kiistanalainen. Sen käsite sisältää kokonaisuudessaan jatkuvan integraation (Continuous Integration, CI) yhdistettynä sekä kääntämisautomaatioon että konfiguraatiohallintaan. Yritykset käyttävät jatkuvaa toimitusta ja sen työkaluja päivittäessään sovelluksia ja ohjelmistotuotteita luodakseen saumattoman käyttäjäkokemuksen. [6]

Jatkuvan toimituksen toiminta perustuu testitapausten, hyväksymisien ja julkaisujen ympärille. Toimitus jakaantuu useaan eri vaiheeseen kuvan 2.1 mukaisesti. Ohjelmakoodin kääntäminen voidaan suorittaa jokaisen kehityksestä johtuneen muutoksen yhteydessä, jonka jälkeen suoritetaan automatisoitua testausta. Testien tulokset palautetaan kehitystiimille, joka puolestaan voi julkaista käännöksen tuotantoon. Muutokset voidaan julkaista automaattisesti hyödyntäen jatkuvaa käyttöönottoa (continuous deployment), mutta muutokset voidaan myös julkaista manuaalisesti, mikä on tavanomaista jatkuvassa toimituksessa. [6]

Jatkuvan toimituksen avulla muutokset ohjelmistossa voidaan toimittaa tuotantoon tai suoraan loppukäyttäjille nopeasti ja turvallisesti. Muutokset voivat olla esimerkiksi uusia ominaisuuksia, konfiguraatiomuutoksia, ohjelmointivirheiden korjauksia tai kokeiluja. Sen tavoitteena on muodostaa toimituksesta rutiininomainen toimenpide, joka voidaan suorittaa aina tarpeen mukaan. Toimituksia voidaan suorittaa muun muassa laajan mittakaavan hajautetuissa järjestelmissä, monimutkaisissa tuotantoympäristöissä ja sulautetuissa järjestelmissä. Ohjelmakoodi on aina valmiina toimitukseen, vaikka useat kehittäjät muokkaisivatkin sitä säännöllisesti. Perinteisessä ohjelmistotoimituksessa koodi-integraatiot, testaamiset ja ohjelmointivirheiden korjaamiset saattavat viedä viikkoja tai jopa kuukausia. Kun tiimit työskentelevät yhdessä automatisoidakseen kääntämisen, tuottamisen ja testaamisen, kehittäjät voivat sisällyttää integraation ja testaamisen päivittäiseen työskentelyynsä. Täten voidaan myös vältellä suurta määrää toisteista työtä. [49]

Jatkuvan toimituksen ansiosta ohjelmistokehitystiimeille jää enemmän aikaa keskittyä käyttäjätutkimukseen ja korkeamman tason testamiseen, kuten tutkivaan testaamiseen, käytettävyystestaamiseen, kuormitustestaamiseen ja turvallisuustestaamiseen. Kyseisiä toimenpiteitä voidaan suorittaa jatkuvasti toimituksen aikana rakentamalla sille liukuhihna (pipeline). Täten tuotteista ja palveluista voidaan kehittää laadukkaita alusta alkaen. Menestyksekkäillä ohjelmistotuotteilla ja -palveluilla on tapana kehittyä huomattavasti elinaikanaan. Panostamalla kääntämiseen, testaamiseen, tuottamiseen ja ympäristön automaatioon voidaan merkittävästi leikata kuluja, jotka menevät ohjelmiston muutoksiin ja toimituksiin. Myös monia julkaisuun liittyviä menoja pystytään täten vähentämään.



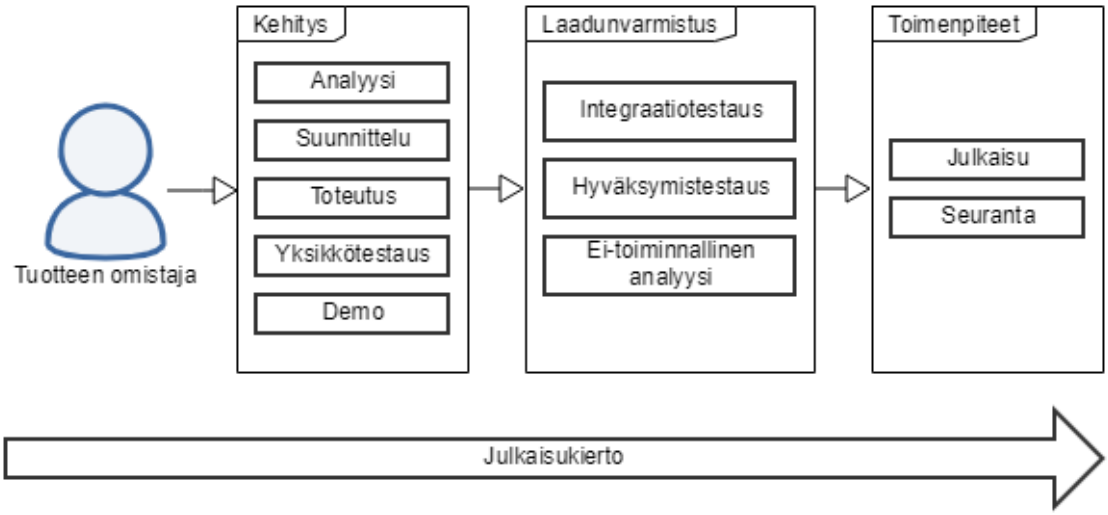
Kuva 2.1. *Jatkuvan toimituksen kulku [53].*

Julkaisemalla tiheämpään tahtiin tiimit voivat olla aktiivisemmin asiakkaiden kanssa tekemisissä, havaita toimivat ratkaisut ja nähdä suoralta kädeltä työnsä lopputulokset. [49]

Verrattuna perinteiseen toimitusprosessiin, jatkuvassa toimituksessa ohjelmisto voidaan toimittaa pienemmällä määrällä ohjelmointivirheitä ja matalammalla riskillä. Kun julkaistaan pieniä muutoksia tiheään tahtiin, virheitä voidaan havaita jo kehitysvaiheessa. Kun automatisoitua testaamista hyödynnetään jokaisella kehityksen tasolla, koodi ei etene myöhempiin vaiheisiin testien epäonnistuessa. Lisäksi pienien muutosten palauttaminen edeltävään tilaan on helppoa. Uusia ominaisuuksia saadaan julkaistua tiheämpään tahtiin kaupalliseen käyttöön, jolloin myös palautetta saadaan tiheämpään tahtiin, josta voidaan ottaa opiksi seuraavia iteraatioita ajatellen. Asiakkaiden värväämineen kehityskumppaneiksi antaa heille osaomistajuuden ja uskollisuuden tunteen, ja kaupallisiin vaatimuksiin pystytään vastaamaan nopeammin. Jatkuvan toimituksen ansiosta vastuu leviää laajasti, jolloin yhteistyö on helpompaa. Se poistaa myös paljon paineita ohjelmiston julkaisuista. Pienien muutosten julkaisun ansiosta kaikki tottuvat säännölliseen julkaisutahtiin, jolloin aikaa jää enemmän suunnittelemiseen ja kehittämiseen. [19]

2.1 Perinteinen toimitusprosessi

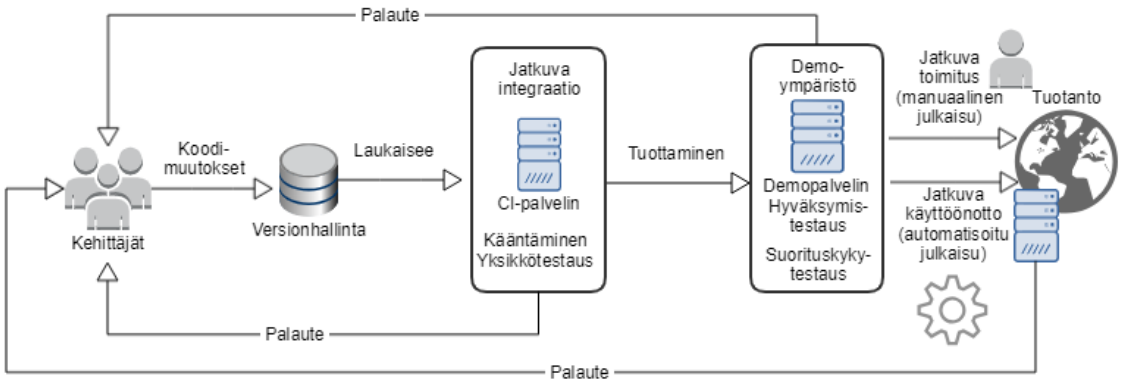
Perinteistä toimitusprosessia on nimensä mukaisesti hyödynnetty jo vuosia, ja suuri osa ohjelmistoalan yrityksistä soveltaa sitä edelleenkin. Toimitusprosessit alkavat asiakkaan vaatimusmäärittelyillä ja päättyvät tuotteen julkaisuun. Kuvassa 2.2 on havainnollistettu perinteisen toimitusprosessin julkaisukierto. Julkaisukierto alkaa siitä, kun asiakasta edustava tuotteen omistaja (product owner) laatii vaatimukset toteutettavalle järjestelmälle. Sen jälkeen työ liikkuu kolmen eri vaiheen välillä, jotka ovat kehitys-, laadunvarmistus- (Quality Assurance, QA) sekä toimenpidevaihe. Kehitysvaiheessa ohjelmistokehittäjät työstävät tuotetta. Kehittäjät käyttävät usein ketteriä tekniikoita sekä nopeuttaakseen kehitystä että parantaakseen kommunikointia asiakkaan kesken. Demotilaisuuksien pitä-



Kuva 2.2. *Perinteisen toimitusprosessin julkaisukierto [32].*

minen on yleinen tapa hankkia asiakkaalta palautetta nopeasti. Kun toteutus on valmis, koodi viedään laadunvarmistustiimille. Laadunvarmistusvaihe edellyttää pääkehityshaaran koodin jäädyttämistä, eli vaiheen aikana pääkehityshaaraan ei saa tehdä muutoksia. Täten varmistetaan, ettei laadunvarmistuksen aikana tapahtuva rinnakkainen kehittäminen johda testien rikkoutumiseen. Laadunvarmistajat suorittavat kaikenlaista testaamista, kuten integraatiotestausta, hyväksymistestausta sekä ei-toiminnallista testausta. Löydetyt virheet palaavat takaisin kehitystiimille korjattaviksi. Kun tuote on todettu tarpeeksi laadukkaaksi, laadunvarmistustiimi hyväksyy seuraavaan julkaisuun sisältyvät ominaisuudet. Viimeisessä ja usein lyhyimmässä toimenpidevaiheessa suoritetaan julkaisu ja tuotannon käyttäytymistä seurataan. Julkaisukierron pituus on usein viikon ja muutaman kuukauden välillä riippuen kehitettävästä järjestelmästä ja organisaatiosta. [32]

Perinteisellä toimitusprosessilla on lukuisia heikkouksia. Toimitus on hidasta, ja asiakas joutuu odottamaan tuotetta kauan vaatimusten määrittelyn jälkeen, jolloin myös asiakkaalta tulevan palautteen saaminen viivästyy. Palaute viivästyy myös yrityksen sisällä, koska laadunvarmistusvaiheessa voidaan löytää virheitä, jotka ilmaantuivat koodiin esimerkiksi pari kuukautta aiemmin. Siten pienetkin virheet voivat viivästyttää julkaisua huomattavasti. Harvoin tapahtuvat julkaisut eivät rohkaise hyödyntämään automaatiota, jolloin julkaisujen ajankohtia on vaikea ennustaa. Pikaiset korjaustarpeet eivät voi usein jäädä odottamaan koko laadunvarmistusvaiheen suoritusta, jolloin niitä täytyy testata eri tavalla tai jättää kokonaan testaamatta. Lisäksi arvaamattomat julkaisut ovat stressaavia toimihenkilöiden kannalta. Julkaisukierteet ovat myös usein tiukasti aikataulutettuja, mikä lisää kehittäjien ja testaajien stressiä. Työn välittäminen tiimiltä toiselle edustaa vesiputousmallia, jossa työntekijät ovat todennäköisemmin kiinnostuneita vain omasta työosuudestaan koko tuotteesta vastaamisen sijaan. Ongelmatilanteet saattavat tällöin johtaa toisten syyttelyyn yhteistyön sijaan. Tiimit eivät ota kokonaisvaltaista vastuuta, vaan työn



***Kuva 2.3**. Jatkuvan integraation, toimituksen sekä käyttöönoton välinen suhde [40].*

valmiuden määritelmät ovat vaihekohtaisia. Kehittäjät ottavat vastuun ainoastaan vaatimusten toteuttamisesta, laadunvarmistustiimi testaamisesta ja toimihenkilöt palvelun julkaisemisesta ja seurannasta. Työtyytyväisyys voi kärsiä, jos ei halua työskennellä koko aikaa samalla osa-alueella, vaan esimerkiksi haluaa kehittää koodia testaamisen ohella. Kyseisten heikkouksien takia ohjelmistoyrityksillä on noussut tarve hyödyntää ketteriä menetelmiä, kuten esimerkiksi jatkuvaa toimitusta. [32]

2.2 Läheiset menetelmät

On olemassa kaksi menetelmää, jotka ovat keskeisiä jatkuvan toimituksen kannalta: jatkuva integraatio sekä jatkuva käyttöönotto. Jokainen menetelmä pyrkii tekemään sekä ohjelmistokehityksestä että ohjelmistojen julkaisemisesta tehokkaampaa [1]. Termien samankaltaisten nimien johdosta niitä näkee usein käytettävän sekaisin keskenään.

Sekä jatkuvan toimituksen että jatkuvan käyttöönoton tarkat määritelmät jäävät usein vähälle huomiolle kirjallisuudessa ja teollisuuden piirissä. Jatkuva käyttöönotto perustuu siihen, että koodimuutokset julkaistaan automaattisesti tuotantoympäristöön ilman manuaalista vuorovaikutusta. Jatkuvassa toimituksessa puolestaan ihminen tekee päätöksen, milloin tuotantoa varten valmis koodi julkaistaan tuotantoon. Kuvassa 2.3 on esitetty menetelmien välinen suhde sekä sovelluksen tuottaminen eri ympäristöihin [40]. Jatkuva integraatio on osa kumpaakin menetelmää, jonka aikana sovellus käännetään ja yksikkötestataan.

2.2.1 Jatkuva integraatio

Jatkuva integraatio on kehitysmenetelmä, jossa kehittäjät vievät työstämäänsä ohjelmakoodia versionhallintaan (commit) mahdollisimman usein. Sen avulla koodimuutokset saadaan tallennettua keskitetysti, jolloin esimerkiksi virhetilanteissa voidaan palata aiemmin toimineeseen versioon koodista. Automatisoitu kääntäminen tarkistaa aikaansaannoksen, jolloin kehitystiimit voivat huomata ongelmia ajoissa. Mitä tiheämpään tahtiin

koodia viedään versionhallintaan, sitä nopeammin voidaan löytää ja paikantaa virheitä. Siten jää enemmän aikaa uusien ominaisuuksien kehittämiselle. Jatkuvalla integraatiolla on monia hyötyjä: ei ole pitkiä ja raskaita versionhallintaan viemisiä, koodin saatavuus paranee, virheiden etsiminen vie vähemmän aikaa ja ohjelmistoa voidaan tuottaa tiheämpään tahtiin. [12]

Jatkuva integraatio ratkaisee ongelman, jossa ohjelmistokehittäjät toimivat usein eristyksissä toisistaan, mutta heidän täytyy viedä tekemänsä koodimuutokset versionhallintaan. Pitkän aikavälin integraatiot johtavat moniin konflikteihin, vaikeasti löydettäviin ohjelmointivirheisiin, koodausstrategioiden hajaantumiseen sekä turhaan vaivannäköön. [21]

Jatkuvassa integraatiossa ohjelmakoodia haetaan versionhallinasta, jolloin koodia voidaan muokata ja päivittää. Muokkausten päätteeksi koodi viedään takaisin versionhallintaan, jolloin sille voidaan suorittaa automaattisesti testejä ja laadunvarmennusta. Jatkuvalla integraatiolla voidaan minimoida raskaiden versionhallintaan viemisten ja ohjelmistopäivitysten tarvetta. Menetelmä vaatii useita kehittäjiä ollakseen hyödyllinen, mutta sillä saadaan varmistettua, että tärkeät muutokset saadaan integroitua sovelluksiin nopeasti. [6]

Monen kehittäjän aikaansaannosten yhdistämisessä on haasteensa. Ohjelmistot voivat olla monimutkaisia, jolloin pienilläkin muutoksilla voi olla suuria ja arvaamattomia seurauksia. Sen takia kehittäjät työskentelevät usein eri kehityshaaroissa (branch), jolloin pääkehityshaara pysyy vakaana. Haaroilla on kuitenkin taipumus erkaantua toisistaan ajan myötä. Vaikka yhden haaran yhdistäminen päähaaraan ei ole vaivalloista, monen pitkään eristyksessä olleen haaran yhdistäminen päähaaraan vaatii usein monien ristiriitojen ratkaisemisia. Haarojen yhdistäminen vaatii usein sen, ettei tiettyä osaa koodista saa muuttaa ennen julkaisua. Suurempien tiimien kohdalla haarojen integraatio vaatii useita regressiotestejä ja ohjelmointivirheiden korjauksia, jotta päästään varmuuteen järjestelmän toimivuudesta yhdistämisen jälkeen [13]. Regressiotestauksella varmistetaan, etteivät ohjelmakoodimuutokset riko ohjelmiston toimivuutta [51].

Kehitettävät ominaisuudet hajautetaan erillisiksi työtehtäviksi, jotka pyritään suunnittelemaan mahdollisiman pieniksi. Tehtävät jaetaan kehittäjien kesken, ja tehtävän valmistuessa kehittäjä vie muutoksen kehityshaaraan. Täten muutokset saadaan nopeasti arvioitaviksi, testattaviksi ja julkaistaviksi. Pienissä osissa työskentely mahdollistaa myös säännöllisen palautteen saamisen kehittäjiltä, testaajilta sekä asiakkailta, jolloin ongelmien huomaaminen ja korjaaminen on helpompaa. Täten jatkuva integraatio johtaa suurempaan suoritustehoon, vakaisiin järjestelmiin sekä laadukkaampaan ohjelmistoon. [13]

Jokaisen versionhallintaan viemisen yhteydessä ajetaan sarja kattavia, nopeasti suoritettavia yksikkötestejä. Niiden pitäisi olla tarpeeksi kattavia antamaan luotettava kuva ohjelmiston toimivuudesta. Jos yksikkötestien suoritus kestää kauan, kehittäjät eivät halua ajaa niitä säännöllisesti, jolloin niitä on vaikeampi ylläpitää. Jos testit epäonnistuvat, ongelmat korjataan välittömästi. Täten varmistetaan, että ohjelmisto on aina toimintakunnossa, ja ettei kehittäjien haarat eroa liikaa pääkehityshaarasta. Testivetoisella kehityksellä voidaan

luoda ylläpidettäviä yksikkötestejä, sillä testit luodaan ennen varsinaisen koodin kirjoittamista. Siten koodista tulee modulaarisempaa ja testattavampaa. [13]

2.2.2 Jatkuva käyttöönotto

Jatkuva käyttöönotto on vaihtoehtoinen menetelmä jatkuvalle toimitukselle. Jatkuvassa käyttöönotossa minimoidaan se aikaväli, jolla ohjelmakoodia kirjoitetaan ja jolla se päätyy tuotantoon loppukäyttäjille. Siinä hyödynnetään eri välivaiheiden automatisointia siten, että lopulta tuotannossa oleva sovellus saadaan päivitettyä. Jatkuvalla käyttöönotolla voidaan nopeuttaa sijoitetun pääoman tuottoa, koska ominaisuuksia saadaan vietyä nopeammin tuotantoon. Lisäksi asiakkailta saadaan nopeammin palautetta uusista ominaisuuksista. [11]

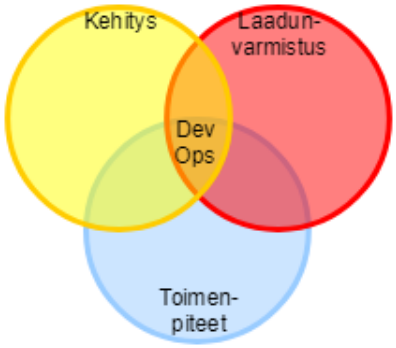
Jatkuvan käyttöönoton työkaluilla tiimi voi tuottaa ohjelmiston välittömästi muutosten yhteydessä sen sijaan, että joutuisi odottamaan tuottamista useiden päivitysten suorittamisten jälkeen. Täten jatkuvalla käyttöönotolla saadaan luotua täysin saumattomia ja jatkuvasti päivittyviä ohjelmistoja [6]. Kääntöpuolena kehitystiimiin vastuulle jää se, kuinka tarkasti hyväksymistestausta sovelletaan. Voidaan esimerkiksi vaatia, että kaikkien hyväksymistestien on onnistuttava, ennen kuin muutokset julkaistaan tuotantoon. Silloin ei ole varmaa, pääsevätkö muutokset todellakin automaattisesti tuotantoon asti, sillä yksikin epäonnistunut testi estää sen. Toisaalta liian löyhillä vaatimuksilla testaustulosten suhteen tuotantoon voi päätyä ohjelmisto, jossa on enemmän virheitä ja puutteita.

Jatkuvassa käyttöönotossa ohjelmistoja pystytään päivittämään uusien ominaisuuksien valmistuttua. Palveluita voidaan skaalata reaaliaikaisesti kuormituksen mukaan, jolloin sekä laite- että ohjelmistovirheistä voidaan toipua mahdollisimman nopeasti. Työketju kehittämisestä liikevoittoon asti pyritään vakiinnuttamaan ja tehostamaan ketterän ohjelmistokehityksen mukaisesti. Täten jatkuvassa käyttöönotossa ominaisuudet julkaistaan suoraan tuotantoon ilman manuaalisia toimenpiteitä, jolloin voidaan välttyä niistä aiheutuvilta virheiltä ja viivästyksiltä. [22]

2.3 DevOps

Ohjelmistokehityksen alkuaikoina ei ole ollut selkeää jaottelua kehityksen, laadunvarmistuksen ja toimenpiteiden välillä. Sama henkilö saattoi kehittää koodia, testata sitä ja julkaista sen tuotantoon. Järjestelmien ja kehitystiimien kokojen kasvaessa kehittäjät ovat alkaneet erikoistumaan tiettyihin osa-alueisiin, koska se on tehokkaampaa tuottavuuden kannalta. Haittapuolena osapuolten välinen kommunikaatio ja yhteistyö heikkenee, etenkin jos osapuolet työskentelevät eri sijanneissa. Sellainen organisaatiorakenne ei ole hyväksi jatkuvassa toimituksessa, joka perustuu osa-alueiden tiiviiseen yhteensovittamiseen. Täten jatkuva toimitus vaatii ympärilleen sopivan kulttuurin toimiakseen täysin. [32]

DevOps-kulttuuri (Development and Operations, suom. kehitys ja toimenpiteet) on yhdistelmä sekä ohjelmistokehittämistä että teknologiapainotteisia toimenpiteitä, jossa työnte-



***Kuva 2.4**. Osa-alueiden välinen suhde DevOps-kulttuurissa [32].*

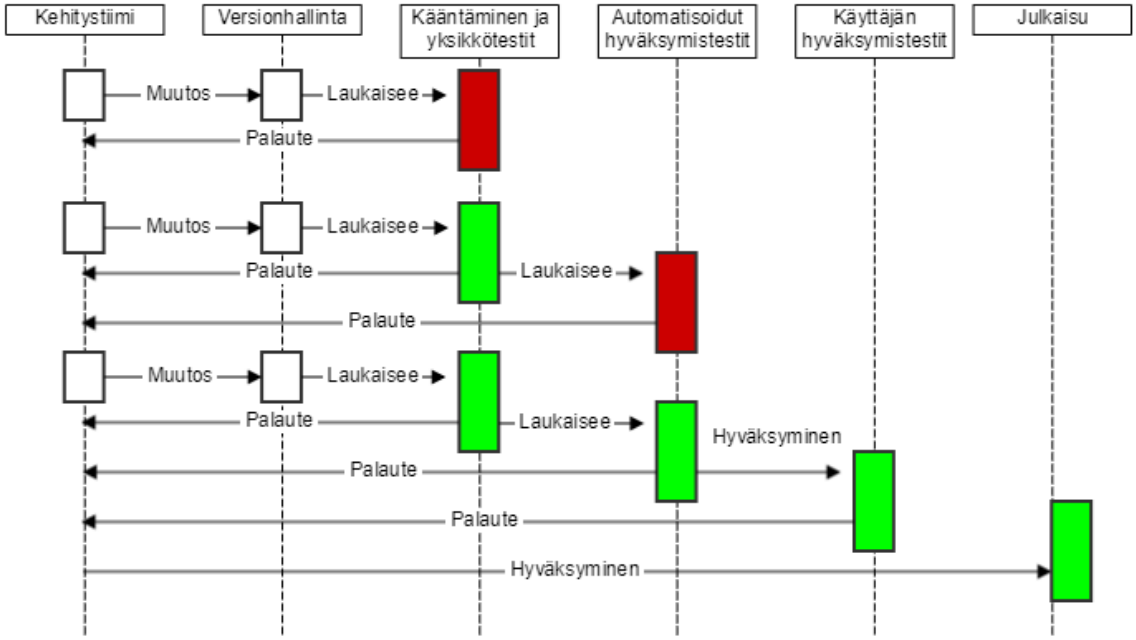
kijät ovat kokonaisvaltaisessa vastuussa kehittämisestä, laadunvarmistuksesta sekä erillisistä toimenpiteistä. Kuva 2.4 havainnolistaa DevOps-kulttuurin sijoittumista osa-alueiden välille. Ketterissä menetelmissä hyödynnettävän automaation ansiosta on mahdollista noudattaa raskaalta tuntuvaa DevOps-mallia siten, ettei tuottavuus kärsi. Useimmat laadunvarmistus- ja toimenpidetehtävät hoituvat liukuhihnalla, jota kehitystiimin jäsenet voivat ohjata. [32]

DevOps-malliin siirtyminen edellyttää muutosta työkulttuurissa ja asenteissa. Tavoitteena on poistaa kehityksen, laadunvarmistuksen ja muiden toimenpiteiden välisiä rajoja. Osapuolten täytyy tehdä yhteistyötä optimoidakseen kehityksen tuottavuutta ja toimenpiteiden luotettavuutta. Työntekjän rooli tiimissä ei saa asettaa rajoitteita tuotteesta vastaamiselle kokonaisuudessaan [50]. Kulttuurista toiseen siirtyminen projektin sisällä on sitä haastavampaa, mitä kauemmin projektia on kehitetty ja mitä suurempi se on. Siitä huolimatta jatkuvan toimituksen hyödyllisyys on suoraan verrannollinen DevOps-kulttuurin käyttöasteelle.

2.4 Liukuhihna

Jatkuvan toimituksen keskipisteenä on toimituksen liukuhihna, josta ilmenee, kuinka kehitystiimi toimittaa muutoksia tuotantoon. Liukuhihnan ansiosta tuotteista voidaan kehittää laadukkaampia [7]. Kuvassa 2.5 on esitetty esimerkki tavanomaisesta jatkuvan toimituksen liukuhihnan suorituksesta. Liukuhihna tarjoaa kehitystiimille reaaliaikaista tietoa vaiheiden suorittamisesta sekä niiden tuloksista. Liukuhihnalla edetään automaattisesti seuraaviin vaiheisiin tehtävien onnistuessa, mutta epäonnistumisien yhteydessä liukuhihnan suoritus keskeytyy.

Jokainen perinteisen toimitusprosessin vaihe on tärkeä ohjelmiston toimittamisen kannalta. Siirryttäessä perinteisestä toimituksesta jatkuvaan toimitukseen kyseisten vaiheiden on myös siirryttävä, mutta niitä pitää pystyä suorittamaan liukuhihnalla automatisoidusti. Automatisoidun tuotannon liukuhihna koostuu jatkuvasta integraatiosta, automatisoiduista hyväksymistestauksesta sekä konfiguraatiohallinnasta. Liukuhihna on sarja skriptejä,



Kuva 2.5. *Jatkuvan toimituksen liukuhihnan suoritus [14].*

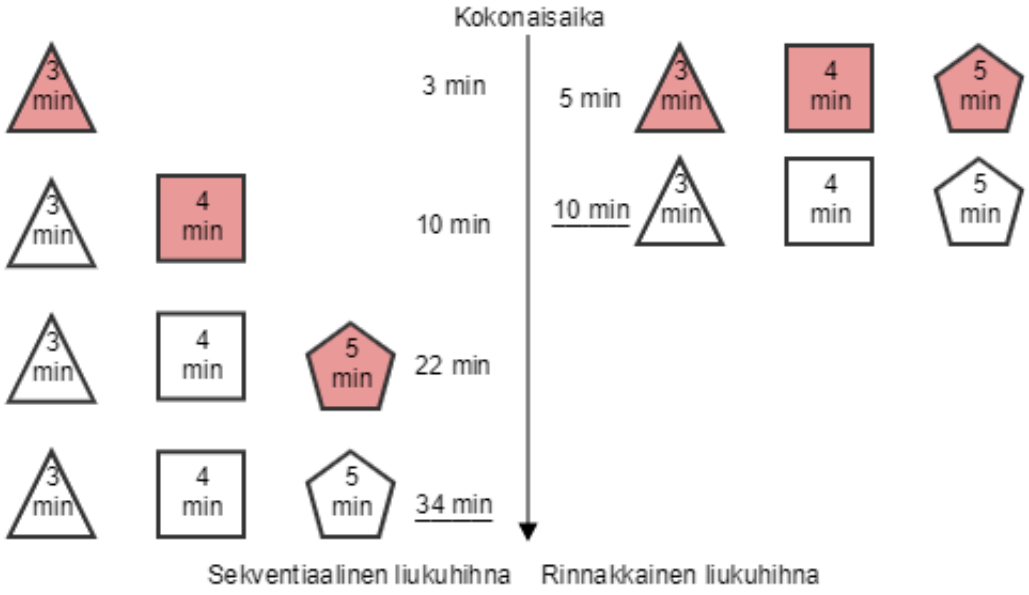
jotka suoritetaan jokaisen versionhallintaan viedyn muutoksen yhteydessä. [32]

Toisistaan riippumattomien tehtävien ajamisessa kannattaa hyödyntää rinnakkaisuutta, jolloin aikaa kuluu vähemmän ja pullonkaulojen määrä vähenee [7]. Kuva 2.6 havainnollistaa, kuinka paljon tehtävien rinnakkaisuus säästää aikaa. Siinä kuviot edustavat liukuhihnassa suoritettavia tehtäviä, ja punainen tausta tarkoittaa epäonnistunutta tehtävää. Kuvassa jokainen tehtävä epäonnistuu ensimmäisellä kerralla johonkin virheeseen, jonka jälkeen se korjataan ja tehtävä onnistuu jatkossa. Sekventiaalinen liukuhihna joudutaan suorittamaan jokaisen epäonnistuneet tehtävän yhteydessä, mutta rinnakkaisessa liukuhihnassa kaikki tehtävät ajetaan, vaikka osa niistä epäonnistuisikin. Lisäksi rinnakkaisessa liukuhihnassa pullonkaulan muodostaa pisin tehtävä, kun taas sekventiaalisessa liukuhihnassa pullonkaula muodostuu tehtäviin kuluvien aikojen summasta.

Tehtävistä tulevan palautteen tulee olla selkeää. Tehtävä voi koostua monista toimenpiteistä, jolloin voi olla vaikeaa huomata, mihin toimenpiteeseen tehtävä kaatui. Virheiden etsintä helpottuu, kun käytetään pieniä välivaiheita ja selkeitä virheviestejä. Liukuhihnalla pitäisi myös pystyä toimittamaan jokin vanhempi versio sovelluksesta. Yksi suurimmista ongelmista palatessa vanhaan versioon on tietojen eheyden varmistaminen. [7]

2.5 Ympäristöt

Ideaalinen työnkulku sisältää ympäristöt kehittämiselle, testaamiselle sekä tuotannolle. Kehittäjät työskentelevät paikallisessa kehitysympäristössä, mikä toimii niin sanottuna hiekkalaatikkona. Siellä voidaan kehittää ohjelmakoodia siihen asti, kunnes kehitettävä

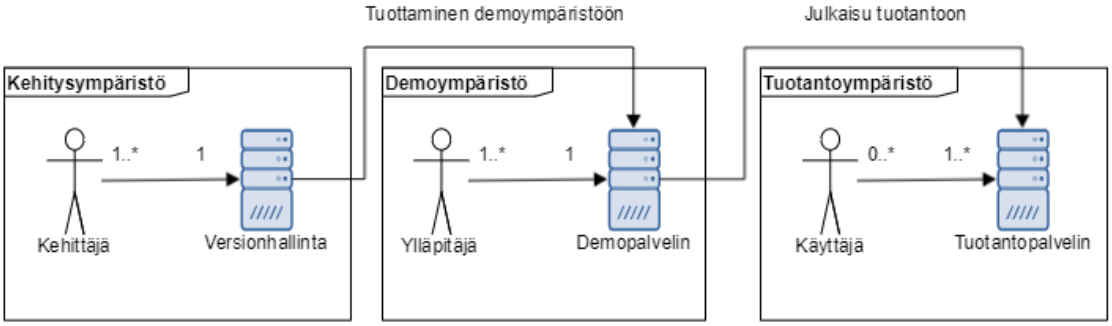


Kuva 2.6. *Sekventiaalisen ja rinnakkaisen liukuhihnan ajankäytön vertailu, jossa kuviot edustavat tehtäviä [7].*

ominaisuus vastaa tarpeisiin. Versionhallintajärjestelmän käyttö kehitysympäristössä on tärkeää, sillä se mahdollistaa muun muassa kehittämisen yhteistyössä muiden kanssa sekä versioiden tallentamisen. Kun ominaisuus on saatu kehitettyä, ohjelmakoodi viedään demoympäristöön (staging environment). [29]

Demoympäristö on osa työnkulkua useimmissa kehitys- ja tuotantoprosesseissa. Siellä voidaan tarkastella etukäteen kehittäjien työstämää ohjelmistoa. Sen tehtävänä on matkia tuotantoympäristöä lähes täydellisesti. Yleisesti demoympäristössä suoritetaan laadunvarmistusta testaamalla ohjelmakoodia, käännöksiä ja päivityksiä. Demoympäristöt vaativat usein täsmälliset kopiot laitteiston, palvelimien, tietokantojen ja välimuistien konfiguraatioista toimiakseen oikein. Demoympäristössä testataan lähes tuotantotasoista sovellusta, jolloin saadaan parempi käsitys siitä, miten sovellus ilmenisi tuotannossa. Ympäristössä suoritetaan testejä, joilla ehkäistään ennalta sekä ongelmia tuotannossa että matalaa suorituskykyä loppukäyttäjillä. Täten voidaan minimoida tuotantoon asti pääsevien virheiden määrä. [39]

Demoympäristö on turvallinen paikka testata muutoksia, joita aiotaan toteuttaa tietoturvakriittisessä ympäristössä. Täten voidaan vähentää odottamattomien virheiden määrää tuotantoympäristössä. Demoympäristö toimii siltana kehittämisen ja tuotannon välillä, jossa suoritetaan sekä laadunvarmistusta että testausta mahdollisimman kattavasti. Demoympäristö voidaan pitää ainoastaan yrityksen sisäisessä käytössä tai se voidaan antaa myös ulkopuolisten saataville. Täten ohjelmistoa voidaan tarkastella ja testata turvallisesti rikkomatta tuotantoympäristöä. Kun demoympäristössä ollaan tyytyväisiä tuotteen ulkoasuun, suoritukseen, luotettavuuteen sekä toiminnallisuuteen, tuote voidaan viedä tuotantoympä-



Kuva 2.7. *Jatkuvassa toimituksessa usein käytettävät ympäristöt kehittämiselle, testaamiselle ja tuotannolle.*

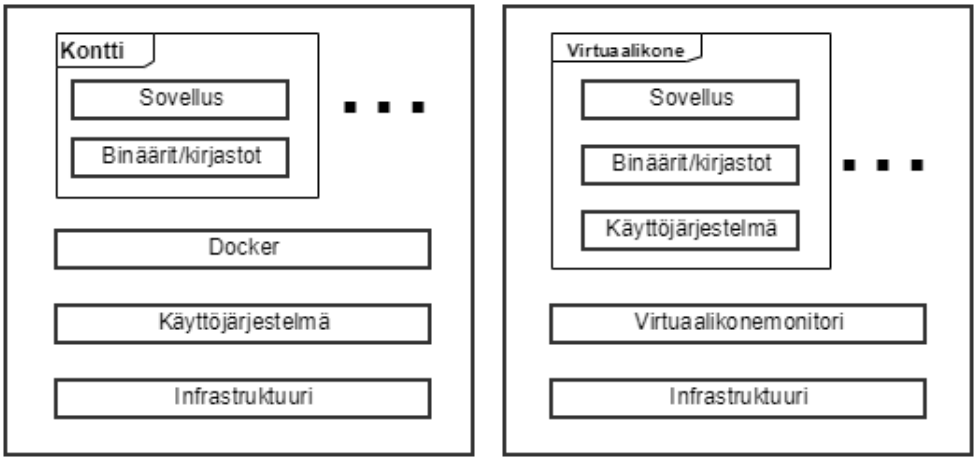
ristöön julkisesti saataville. [29]

Kuvan 2.7 mukaisesti kehittäjät vievät kehittämäänsä koodia versionhallintaan, ja esimerkiksi jokaisen viemisen yhteydessä voidaan tuottaa uusi demo muutosten pohjalta. Demoympäristössä suoritetaan automatisoitua testaamista, mutta myös manuaalista testausta voidaan suorittaa tarvittaessa. Lopuksi joku ylläpitäjistä joko hyväksyy tai hylkää tehdyt muutokset hyödyntäen päätöksenteossaan testituloksia. Mikäli muutokset hyväksytään, ne julkaistaan yhdelle tai useammalle tuotantopalvelimelle loppukäyttäjien saataville.

2.6 Virtualisointi ja kontitus

Jatkuvassa toimituksessa on usein tarvetta eriyttää ympäristöt ja mahdollisesti niiden sisältämät toimeenpiteet toisistaan riippumattomiksi kokonaisuuksiksi. Eriyttämisen ansiosta sovelluksia voidaan ajaa omissa kokonaisuuksissaan, jolloin ne käyttävät ainoastaan niille annettuja resursseja. Täten käytettäviä resursseja ja koko ympäristöä voidaan hallita paremmin [34]. Eriyttäminen on mahdollista sekä perinteisellä virtualisoinnilla että suosiotaan kasvattaneella kontituksella (containerization). Kuvassa 2.8 on havainnollistettu ero kontituksen ja virtualisoinnin välillä.

Virtualisointi perustuu virtuaalikoneiden hyödyntämiselle eri käyttöjärjestelmien ollessa eriytetty toisistaan. Virtuaalikone emuloi tietokonearkkitehtuuria ja tarjoaa fyysisen koneen toiminnallisuuksia. Sovellusten eriyttäminen voidaan saavuttaa ajamalla sovelluksia eri virtuaalikonekuvina (virtual machine image). Jokainen sovellus käynnistetään erillisenä kuvana riippuvuuksineen ja käyttöjärjestelmineen, joita virtuaalikone suorittaa. Monet työkalut tukevat virtualisoinnin avulla tapahtuvaa tuotantoa, ja virtualisoinnilla voidaan kätevästi eriyttää kehitys- ja testausympäristöt toisistaan. Virtualisoinnilla on kuitenkin kolme huomattavaa heikkoutta. Virtuaalikone emuloi kokonaista tietokonearkkitehtuuria pyörittääkseen vieraskäyttöjärjestelmää, joka voi johtaa heikkoon suorituskykyyn sekä turhaan kuormitukseen jokaisen toimenpiteen yhteydessä. Emulaatio vaatii paljon resursseja ja jokainen sovellus täytyy erikseen emuloida, minkä takia tavanomaisilla työpöy-



***Kuva 2.8**. Konttien ja virtuaalikoneiden välinen ero [18].*

täkoneilla suoritetaan vain vähän sovelluksia samanaikaisesti. Jokainen sovellus toimitetaan kokonaisen käyttöjärjestelmän kanssa, jolloin virtuaalikonekuvien koot ovat suuria ja tuottamisen yhteydessä täytyy lähettää ja tallentaa suuri määrä tietoa. [32]

Kontitus tarjoaa erilaisen ratkaisun sovellusten eriyttämiselle. Jokainen sovellus toimitetaan riippuvuuksineen, mutta ilman käyttöjärjestelmää. Sovellukset ovat vuorovaikutuksessa suoraan isäntäkäyttöjärjestelmän kanssa, jolloin ei tarvita erillisiä vieraskäyttöjärjestelmiä. Kontitus tarjoaa paremman suorituskyvyn ilman resurssien tuhlaamista [32]. Kontitukseen on tarjolla ilmainen työkalu nimeltään Docker, jota useimmat jatkuvan toimituksen työkalut pystyvät hyödyntämään. Se tarjoaa kehittäjälle vapauden kääntää, hallita ja turvata sovelluksia siten, etteivät teknologia ja infrastruktuuri aiheuta rajoitteita. Dockerin ansiosta sekä perinteisiä että pilvessä toimivia sovelluksia voidaan viedä automatisoituun ja turvalliseen toimitusketjuun. Docker kasvattaa tuottavuutta ja lyhentää aikaa, joka kuluu sovellusten viemiseen kaupalliseen käyttöön. [16]

Dockerissa kontitus tapahtuu konttien (container) avulla. Kontti on ajonaikainen ilmentymä Docker-kuvasta (image), joka puolestaan on suoritettava paketti, joka sisältää kaiken tarvittavan sovelluksen ajamiseen, kuten ohjelmakoodin, kirjastoja, ympäristömuuttujia sekä konfiguraatiotiedostoja. Konttien käyttämisellä on monia hyötyjä. Niiden joustavuuden ansiosta monimutkaisetkin sovellukset pystytään viemään kontteihin. Kontit ovat kevyitä, sillä ne käyttävät isäntäkäyttöjärjestelmän ydintä jaetusti. Päivityksiä voidaan tuottaa lennosta, ja kontissa oleva sovellus voidaan kääntää paikallisesti, tuottaa pilveen ja käyttää missä tahansa. Lisäksi identtisiä kontteja voidaan luoda ja jakaa automaattisesti. Kontti ajetaan erillisenä prosessina, kun taas virtuaalikoneen täytyy aina sisältää erillinen käyttöjärjestelmä. Siten virtuaalikone tarjoaa ympäristön, joka sisältää sovelluksen kannalta tarpeettomia resursseja, mikä tekee siitä raskaamman. [18]

Kontit tukevat sekä jatkuvaa integraatiota että toimitusta sallimalla kehittäjien kääntää koodia yhteistyössä jakamalla keskenään Docker-kuvia. Täten ohjelmiston tuottaminen

eri infrastruktuureihin yksinkertaistuu. Dockerilla pystytään luomaan käytäntö, joka integroituu käytettäviin jatkuvan toimituksen työkaluihin, eri käyttöjärjestelmien ympäristöihin sekä paikallisiin että pilvi-infrastruktuureihin [17]. Docker-kuvat ovat huomattavasti pienempiä kuin virtuaalikonekuvat. Kontituksessa eriytys tapahtuu samalla tasolla kuin missä isäntäkäyttöjärjestelmän prosessit pyörivät, mutta kontit eivät kuitenkaan jaa riippuvuuksiaan keskenään. Konttien kirjastoilla voi olla tietyt versiot, ja yhden kontin kirjaston päivittäminen ei vaikuta muiden konttien kirjastoihin. Kontituksen aikaansaamiseksi Docker luo sarjan Linux-pohjaisia nimiavaruuksia (namespace) ja hallintajoukkoja kontille. Täten Dockerin turvallisuus perustuu Linuxin käyttöjärjestelmäytimen (kernel) tarjoamaan prosessien eriytykseen. [32]

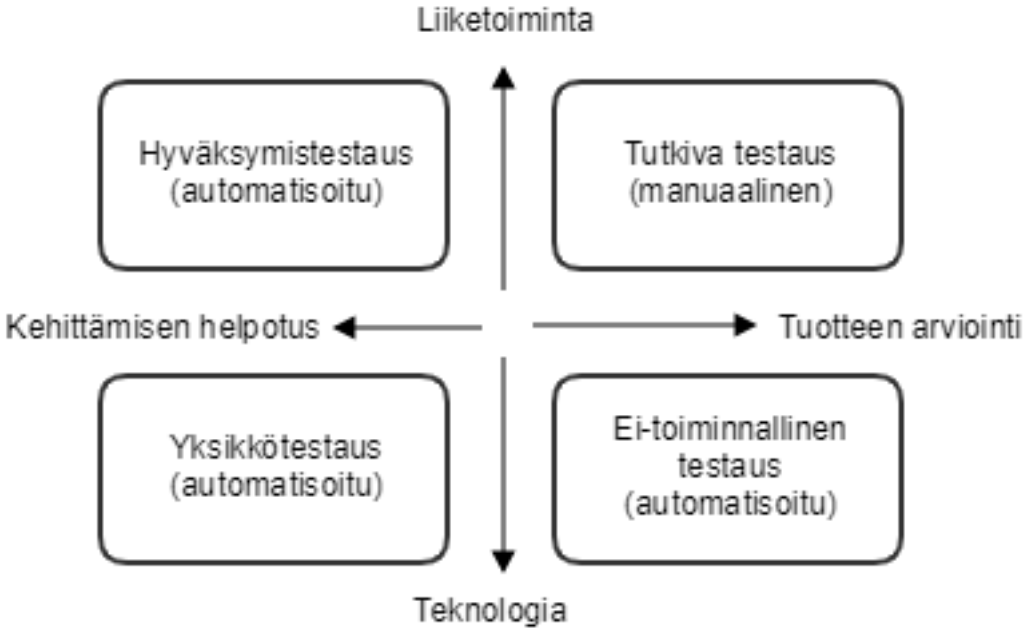
2.7 Testaus

Laadukkaan ohjelmiston rakentaminen perustuu siihen, että testatuista koodimuutoksista saadaan vastaanotettua palautettua nopeasti. Perinteisesti koodimuutoksia on tutkittu ja testattu manuaalisesti laajassa mittakaavassa. Testaajien on täytynyt seurata dokumentaatiota siitä, kuinka järjestelmän eri toimintoja testataan varmistaakseen järjestelmän oikeellisuuden, ja testaaminen on aloitettu vasta pitkäjaksoisen kehittämisen jälkeen. [14]

Manuaalinen regressiotestaus vie paljon aikaa ja on suhteellisen kallista, joten se luo pullonkaulan ohjelmiston tiheätahtiselle tuottamiselle. Siten myös palautteen saaminen kestää kauemmin. Manuaalinen testaus ja tarkasteleminen ei ole luotettavaa, koska ihmiset soveltuvat tietokonetta heikommin toisteisten tehtävien kuten regressiotestien suorittamiseen. Lisäksi pelkästään koodia tarkastelemalla on erittäin vaikeaa ennustaa muutosten suuruutta monimutkaisissa ohjelmistoissa. Järjestelmillä on myös taipumusta kehittyä, joten testausdokumentaation pitäminen ajan tasalla vaatii runsasta vaivannäköä. Laadukkaan ohjelmiston tuottamiseen on käytettävä erilaista lähestymistapaa. Tavoitteena on ajaa useita erityyppisiä testejä jatkuvan toimituksen läpi, niin automaattisia kuin manuaalisiakin testejä. [14]

Automatisoidun hyväksymistestauksen vaihe sisältää joukon asiakkaan ja laadunvarmistustiimin laatimia testejä. Vaiheen aikana suoritettujen testien perusteella päätetään, onko tuote valmis julkaistavaksi. Jos yksikin hyväksymistesti epäonnistuu, liukuhihnan suoritus pysäytetään ja tuote ei etene julkaistavaksi. Vaiheen tavoitteena on tehdä tuotteesta laadukas sen sijaan, että sen toimivuutta pitäisi varmentaa jälkeenpäin. Kun kehittäjä saa toteutuksen valmiiksi, ohjelmisto toimitetaan hyväksymistestattuna, jolloin ohjelmiston varmistetaan olevan asiakkaan toiveiden mukainen. Hyväksymistestaus vaatii usein tiivistä yhteistyötä asiakkaan kanssa, ja testejä kannattaa luoda jo toimituksen alussa. Automaattisten hyväksymistestien hyödyntäminen vanhoissa järjestelmissä voi olla haastavaa. [32]

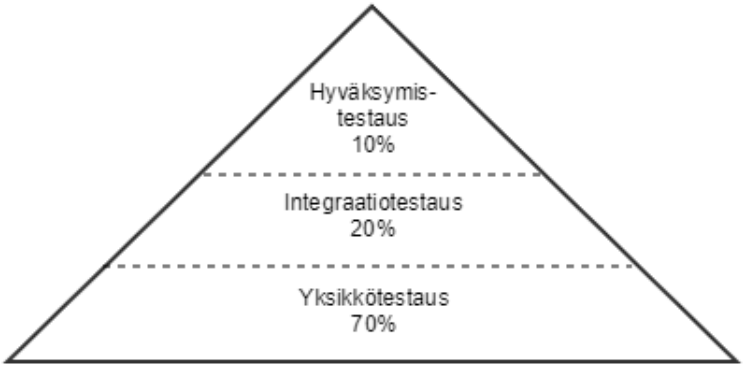
Kuvassa 2.9 testaustavat on jaettu neljään kategoriaan: hyväksymistestaus, tutkiva testaus, yksikkötestaus sekä ei-toiminnallinen testaus. Kuva havainnollistaa, kuinka eri testausta-



Kuva 2.9. *Testaustapoja, joita voidaan hyödyntää jatkuvassa toimituksessa [32].*

vat painottuvat liiketoiminnan ja teknologian välillä, ja kuinka ne helpottavat sekä kehittämistä että tuotteen arviointia. Automatisoidut hyväksymistestit edustavat toiminnallisia vaatimuksia liiketoiminnan näkökulmasta. Asiakkaat määrittelevät niitä usein kertomusten ja esimerkkien avulla, joita kehittäjät hyödyntävät ohjelmiston suunnittelussa. Yksikkötestit ovat automatisoituja testejä, joiden avulla testataan useita pieniä kokonaisuuksia ohjelmakoodissa. Tutkivassa testauksessa järjestelmä yritetään rikkoa tai sitä yritetään kehittää manuaalisesti. Ei-toiminnallisilla testeillä mitataan järjestelmän ominaisuuksia, kuten suorituskykyä, skaalautuvuutta, turvallisuutta ja niin edelleen. Järjestelmän käytettävyyttä voidaan testata manuaalisesti. Laadunvarmistuksella ei ole erityistä paikkaa toimitusprosessissa, vaan se on pikemminkin rooli kehitystiimissä. Manuaalisessa laadunvarmistuksessa ei ole enää tarvetta suorittaa toisteisia tehtäviä, vaan ne käsitellään automatisoidusti. Kategorisointi ei sisältänyt integraatiotestejä, koska niiden määritelmä on kontekstiriippuvainen. Mikropalveluarkkitehtuureissa ne vastaavat usein hyväksymistestausta, koska palvelut ovat pieniä ja tarvitsevat ainoastaan yksikkö- ja hyväksymistestejä. Modulaarisessa sovelluksessa integraatiotestit ovat rajapintatestejä, jotka sitovat moduuleita ja testaavat niitä yhdessä. [32]

Testausstrategiaa laatiessa on tärkeää priorisoida eri testausmenetelmät omilla painoarvoillaan. Kuvan 2.10 pyramidissa on esitetty yleisesti sopivina pidetyt painoarvot testausmenetelmille, jossa yksikkötestausta pidetään huomattavasti tärkeimpänä menetelmänä. Yksikkötestit pystyvät kattamaan laajasti ohjelmakoodin eri suorituspolkuja ja niitä pystytään suorittamaan nopeasti. Integraatiotestit ovat hitaampia suorittaa ja niitä on vaikeampi ylläpitää. Lisäksi virheet voivat oman koodin sijaan olla peräisin integroitavas-



Kuva 2.10. *Testausmenetelmien painoarvot pyramidissa esitettynä [15].*

ta kohteesta, jolloin virheiden löytäminen on vaikeaa. Hyväksymistestien kirjoittaminen, suorittaminen ja ylläpito syö paljon resursseja, joten niiden määrä kannattaa minimoida. Vaikka ne antavat hyvän kuvan järjestelmän toimivuudesta, niillä on raskasta testata kattavasti järjestelmän toimivuutta eri tilanteissa. Yleisesti ne soveltuvat parhaiten eri käytännön tilanneiden testaamiseen. Esimerkiksi käyttäjän sisäänkirjautuminen verkkokauppaan, tuotteen lisääminen sekä poistaminen ostoskorista ja uloskirjautuminen voisi olla yksi käyttötapaus, jota voitaisiin hyväksymistestata. [15]

3. JATKUVAN TOIMITUKSEN TYÖKALUT

Jatkuva toimitus vaatii sopivat työkalut prosessin suorittamiseksi. Täten ennen työkalun hankkimista kannattaa miettiä, millaisia ominaisuuksia työkalusta pitäisi löytyä. Työkaluilla varmistetaan asianmukainen kommunikaatio ohjelmistotiimin kehittäjien, testaajien, laadunvarmistajien ja ylläpitäjien kesken [42]. Jatkuvan toimituksen työkaluilla on yhteistä se, että toimenpiteet perustuvat pelkästään koodin ympärille. Siten kaikki prosessin vaiheet voidaan automatisoida.

Jatkuvan toimituksen työkalua hyödyntämällä tuotetta voidaan jatkuvasti päivittää testausvaiheen jälkeen. Tuotteen ominaisuuden päivittämisen jälkeen on tärkeää seurata kyseistä ominaisuutta ja varmistaa, että se toimii oikein. Seurantatyökaluilla voidaan havaita virheelllisiä muokkauksia ja päivityksiä, ennen kuin ne vaikuttavat asiakkaan toimintaan. Siten voidaan parantaa asiakastyytyväisyyttä ja pienentää ongelmien ilmenemisen todennäköisyyttä. Jatkuvan toimituksen ohjelmistoilla on kyky muuttaa tuotetta äärimmäisen paljon. Sen takia on tärkeää, että valittava ohjelmisto on turvallinen ja että vahingossa tehdyt muutokset eivät pääse tuotantoon. Ohjelmistolla tulisi olla pääsynhallintamekanismi, jolla valitut henkilöstön jäsenet pääsisivät tekemään tiettyjä muutoksia tuotteeseen. Vahva turvallisuusjärjestelmä on elintärkeä toimituksen hallinnan kannalta. [6]

Jatkuvan toimituksen sijaan on mahdollista käyttää jatkuvaa käyttöönottoa. Prosessien suurimpana erona on se, että jatkuvassa toimituksessa ennen tuotantoon julkaisua ohjelmisto on testattava ja hyväksyttävä manuaalisesti, kun taas jatkuvassa käyttöönotossa tietokone suorittaa hyväksymisen automaattisesti. Manuaalinen testaaminen vie aikaa ja vaivannäköä, joten testaamisessa kannattaa hyödyntää työkalua, joka suorittaa testit automaattisesti. Jotkin työkalut tarjoavat erikoistuneita testausominaisuuksia, kuten esimerkiksi asiakaspalauteominaisuuksia, joilla saadaan kartoitettua asiakkaiden näkemyksiä tuotteesta. [6]

Yksi suurimmista kysymyksistä on, haluaako työkalun ohjelmistopalveluna eli SaaS-palveluna (Software as a Service), vai tehdäänkö työkalu itse paikallisesti. SaaS-työkalut huolehtivat itse palvelimiensa ylläpidosta ja antavat käyttäjille enemmän aikaa keskittyä kehittämisen muihin osa-alueihin [6]. Lisäksi pilvessä toimivat SaaS-työkalut päivittyvät ajan myötä tarjoten uusia ominaisuuksia ja tukea ongelmatilanteissa.

Paikallinen, itse tehty työkalu on erittäin joustava laajennettavuuden ja muokattavuuden suhteen. Lisäksi tietoa voidaan käsitellä turvallisesti, koska se pysyy yrityksen sisällä. Kääntöpuolena paikallinen työkalu vaatii enemmän aikaa sen ylläpitoon ja hallintaan. Lisäksi paikallinen työkalu voi olla välttämätön vaihtoehto, jos yrityksen käytäntöjen mukaan käsiteltävien tietojen pitäisi jäädä vain yksityksen sisälle, mahdollisesti vielä palo-

muurin taakse. Yrityksillä ei ole välttämättä suurta budjettia jatkuvan toimituksen toteuttamiseen. Palvelu voi erikseen laskuttaa sen ominaisuuksien käyttämisestä, jolloin kannattaa tarkasti suunnitella rahan sijoittaminen eri ominaisuuksien kesken. Hintaan voi vaikuttaa esimerkiksi käännösten määrä päivässä tai muulla aikavälillä, rinnakkaisten käännösten tukeminen, käyttäjien määrä ja tiedon säilytysaika. Paikallisella työkalulla ei ole vastaavia kustannuksia, mutta sen ylläpitäminen syö henkilöstöresursseja. [43]

Jatkuvan toimituksen palvelua valittaessa on tärkeää kiinnittää huomioita siihen, mitä eri palveluita siihen on mahdollista integroida. Useat jatkuvan toimituksen palvelut tarjoavat yhteensopivuuden moniin muihin ohjelmistopalveluihin, kuten sovellusten levitykseen sekä tuotannon että tietokoneiden hallintaan. Kolmannen osapuolen työkalujen hyödyntäminen jatkuvassa toimituksessa mahdollistaa toimituksen virtaviivaistamisen, ja työkaluja pystytään hallitsemaan keskitetysti ainoastaan yhden sovelluksen kautta. [6]

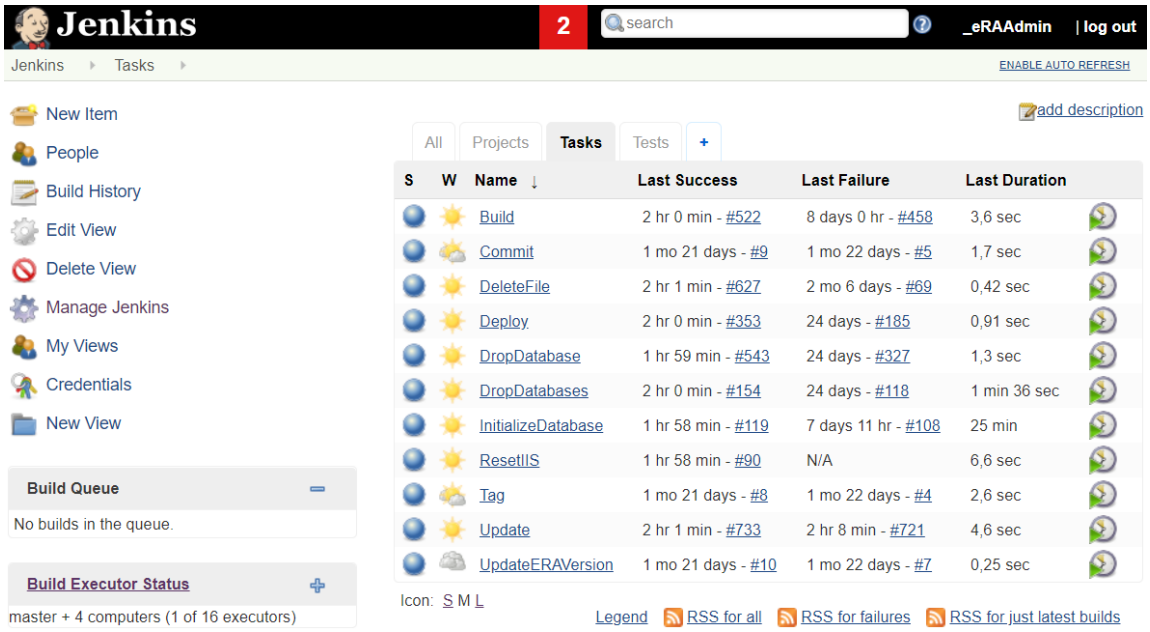
Jatkuvan toimituksen toteuttamiselle on tarjolla valtava määrä kaupallisia työkaluja. Ne ovat keskenään hyvin samankaltaisia, koska niiden tarkoituksena on mahdollistaa jatkuva toimitus ohjelmistoprojekteille. Niistä monet ovat helposti saatavilla, ja useimmat ovat ainakin peruskäytöltään ilmaisia [43]. Työkaluista kuitenkin löytyy eroavaisuuksia muun muassa teknologioiden ja yhteensopivuuksien kanssa, joten niistä kannattaa valita sopivin vaihtoehto ohjelmistoprojektin kannalta. Tässä luvussa tarkastellaan vain huomattavimpia ja käytetyimpiä työkaluja.

3.1 Jenkins

Vuonna 2004 Java-kehittäjä Kohsuke Kawaguchi työskenteli Sun-yrityksellä. Kawaguchi kyllästyi rikkinäisiin käännöksiin ja halusi kehittää tavan, jolla voitaisiin tutkia, toimiiko ohjelmakoodi ennen sen viemistä versionhallintaan. Täten hän rakensi Java-pohjaisen automaatiopalvelimen nimeltään Hudson. Siitä tuli suosittu Sunilla, ja se alkoi leviämään myös muihin yrityksiin avoimena lähdekoodina. Vuonna 2011 Hudsonista haarautui automaatiopalvelu nimeltään Jenkins, josta tuli huomattavasti aktiivisempi ja käytetympi palvelu. [23]

Jenkins on suosituin avoimeen lähdekoodiin perustuva ilmainen automaatiopalvelu. Se tarjoaa satoja liitännäisiä (plugin), joita voidaan hyödyntää ohjelmistojen kääntämisessä, tuottamisessa ja automatisoinnissa. Jenkins soveltuu sekä jatkuvaan integraatioon että jatkuvaan toimitukseen missä tahansa ohjelmistoprojektissa. Se on itsenäinen Java-ohjelma, jota voidaan ajaa suoraan sellaisenaan eri käyttöjärjestelmillä. [25]

Jenkins tarjoaa web-käyttöliittymän, jonka kautta se voidaan alustaa ja konfiguroida. Kuvassa 3.1 on esitetty esimerkki Jenkinsin käyttöliittymästä, josta ilmenee muun muassa määriteltyjä tehtäviä sekä niiden suoritustilastoja. Jenkins on helposti laajennettavissa sen liitännäiskeskeisen arkkitehtuurin ansiosta, joten se tarjoaa rajattomasti käyttömahdollisuuksia. Sillä voidaan jakaa suoritettavaa työkuormaa useiden eri koneiden kesken, ja tietyt koneet voidaan asettaa suorittamaan vain tiettyjä toimenpiteitä. [25]



***Kuva 3.1**. Esimerkkikuva Jenkinsin käyttöliittymästä.*

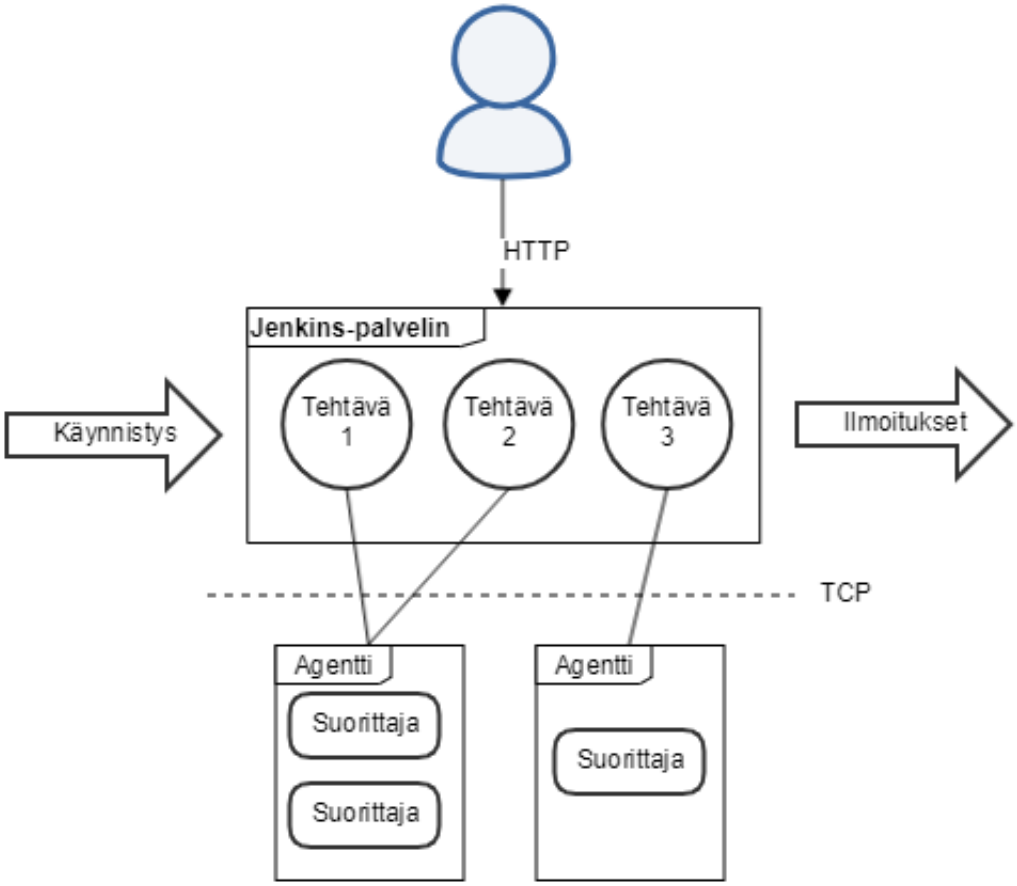
Jenkinsissä on työn kirjoitushetkellä käytettävissä yli 1500 liitännäistä, jotka tukevat kehittämisen automatisointia [28]. Alun perin automaatiota tarjottiin pelkästään Java-koodin jatkuvaan integraatioon sekä toimitukseen, mutta nykyään liitännäisillä voidaan automatisoida muitakin prosesseja. Liitännäiset voidaan jakaa viiteen osa-alueeseen niiden käyttötarkoitusten mukaan, jotka ovat alustat, käyttöliittymä, ylläpito, lähdekoodin hallinta sekä kääntämisen hallinta. [23]

3.1.1 Pääpalvelin ja agentit

Jenkinsillä tehtävien hallinta tapahtuu pääpalvelimella (master). Se on altis kuormitukselle, koska ohjelmistotiimillä tai yrityksellä voi olla tarvetta suorittaa monta tehtävää samanaikaisesti. Sen takia tehtäviä ei pitäisi suorittaa pääpalvelimella, vaan sen kannattaa ohjata tehtävät erillisille agenteille (agent, slave) [32]. Agentti on tyypillisesti tietokone tai kontti, joka ottaa yhteyden pääpalvelimeen ja suorittaa sen antamia tehtäviä [26].

Kuva 3.2 havainnollistaa pääpalvelimen ja agenttien välistä vuorovaikutusta sekä pääpalvelimen vastuualueita. Pääpalvelin vastaanottaa käännösten (build) käynnistyspyyntöjä, jotka voivat syntyä manuaalisesti, ajastetusti tai siinä yhteydessä, kun koodia viedään versionhallintaan. Palvelin tuottaa ilmoituksia käännösten lopputuloksista, jotka voidaan ohjata eri palveluihin, kuten esimerkiksi sähköpostiviesteihin. Palvelin käsittelee ylläpitäjän lähettämiä HTTP-pyyntöjä (Hypertext Transfel Protol, tiedonsiirtoprotokolla), jotta ylläpitäjä voi manipuloida palvelimen toimintaa. Lisäksi palvelin hallitsee käännösympäristöä jakamalla käännöksen aikana tapahtuvien tehtävien suoritusta agenttien kesken. [32]

Agentit ovat konfiguroitavissa monin tavoin. Niille pystyy asettamaan esimerkiksi rin-



Kuva 3.2. *Pääpalvelimen ja agenttien välinen vuorovaikutus [32].*

nakkaisten käännösten maksimimäärän sekä käynnistymistavan. Rinnakkaisten käännösten maksimimäärä määräytyy agentilla sijaitsevien suorittajien (executor) määrän mukaan. Suorittajien avulla agentti voi suorittaa monia tehtäviä rinnakkain [26]. Agentti voidaan käynnistää muun muassa Java Web Start -ohjelmalla. Tällöin agentissa täytyy avata web-käyttöliittymän kautta ladattu tiedosto, joka muodostaa TCP-yhteyden (Transmission Control Protocol, tietoliikenneprotokolla) pääpalvelimeen. Siten agentin ei täydy olla tavoitettavissa pääpalvelimen kautta, vaan ainoastaan agentti pystyy tavoittamaan pääpalvelimen. Pääpalvelimeen pystyy konfiguroimaan portin, minkä kautta se kuuntelee agenttien yhdistämispyyntöjä. Windows-käyttöjärjestelmällä Java Web Start käynnistää Windows-palvelun (Windows service), joka vastaa agentin suorittamisesta. Täten agentti voidaan käynnistää automaattisesti tietokoneen uudelleenkäynnistysten yhteydessä.

Agentti voidaan käynnistää myös komentorivin kautta, jolloin agentti ei tarvitse erillistä Windows-palvelua toimiakseen. Siten agentin graafisesta käyttöliittymästä pystyy näkemään tehtävät, joita pääpalvelimen kautta ollaan määrätty tekemään. Tapahtumien näkeminen voi olla hyödyllistä tai jopa välttämätöntä esimerkiksi käyttöliittymätestejä suoritettaessa.

3.1.2 Tehtävät

Jenkinsissä voidaan konfiguroida tehtäviä (job), jotka soveltuvat yksinkertaisten ja lyhyiden toimenpiteiden suorittamiseen. Tehtävä voidaan määritellä lisäämällä uusi vapaamuotoinen projekti (freestyle project) Jenkinsin käyttöliittymän kautta.

Tehtävillä ja liukuhihnoilla on paljon yhteisiä asetusvaihtoehtoja. Molemmille voidaan esimerkiksi määritellä parametreja, ajastetun käynnistyksen sekä rinnakkaisten ajojen sallimisen. Suurimpana erona tehtäville määritellään käännöskomentoja, jotka suoritetaan tehtävää ajettaessa. Jenkinsissä on oletuksena tuki monille erilaisille komentotyypeille, kuten Windows batch-komennoille, shell-komennoille ja Gradle-skripteille. Komentotyypit ovat laajennettavissa liitännäisten ansiosta. Alla on esimerkki tehtävälle määritellystä batch-komennosta, joka tulostaa tervehdyksen.

```
echo "Hello %name%!"
```

Tehtävälle on määritelty parametri *name*, joka voidaan liukuhihnojen tapaan syöttää ennen ajoa joko Jenkinsin käyttöliittymän kautta tai liukuhihnassa 'build job'-komennon yhteydessä.

3.1.3 Liukuhihnat

Jenkinsiin voidaan viedä Jenkinsfile-nimisiä liukuhihnaskriptejä joko versionhallinnasta tai kirjoittamalla niitä käyttöliittymän kautta. Monet versionhallintajärjestelmät ovat tuettuja liitännäisten ansiosta. Jenkinsin liukuhihnat ovat joko deklaratiivisia tai skriptattuja. Deklaratiivinen liukuhihna on helposti omaksuttava, ja se käyttää Groovy-yhteensopivaa syntaksia. Groovy on Apachen kehittämä ohjelmointikieli Java-alustalle, jonka tavoitteena on parantaa kehittäjien tuottavuutta sen ytimekkään ja helposti lähestyttävän syntaksin ansiosta [20]. Liukuhihnalle määritellään erilaisia rakenteisia lohkoja loogisessa järjestyksessä. Ohjelmassa 3.1 on esitetty yksinkertainen deklaratiivinen liukuhihna, joka ottaa parametrina nimen ja tulostaa tervehdyksen.

Deklaratiivisessa liukuhihnassa kaikki lohkot menevät uloimman *pipeline*-lohkon sisälle. Liukuhihna, tai tietty vaihe (stage), suoritetaan *agent*-lohkon määrittelemällä agentilla. Arvolla *any* voidaan käyttää mitä tahansa vapaana olevaa agenttia. Tehtävä voidaan määrätä suoritettavaksi tietyllä agentilla *labelin* perusteella. Lisäksi agentin pystyy myös asettamaan pyörimään Docker-kontin sisällä. [27]

Liukuhihnassa voi käyttää vapaaehtoista *options*-direktiiviä, jonka sisällä voidaan muokata liukuhihnan asetuksia. Esimerkiksi asetuksella *skipDefaultCheckout* voidaan estää agentin pääsy versionhallintaan [27]. Pipeline-liitännäinen tarjoaa monta asetusta oletuksena, mutta asetuksia saa hankittua lisää muiden liitännäisten kautta.

```
pipeline {
  agent any
  parameters {
    string(description: 'Your name.', name: 'name')
  }
  stages {
    stage('Greet') {
      steps {
          echo "Hello ${params.name}!"
      }
    }
  }
}
```

Ohjelma 3.1. *Deklaratiivinen liukuhihna, joka tervehtii käyttäjää.*

Direktiivin *parameters* sisältävälle liukuhihnalle voidaan syöttää parametreja liukuhihnan laukaisun yhteydessä. Jokainen parametri edustaa tietotyyppiä, kuten esimerkiksi *string* tarkoittaa merkkijonoa ja *booleanParam* totuusarvoa [27]. Parametreilla voidaan vaikuttaa liukuhihnan toimintaan, jolloin liukuhihnaa voidaan suorittaa eri tavoin.

Lohkon *stages* sisällä suoritetaan liukuhihnan toimenpiteet. Lohkon sisälle on tultava vähintään yksi *stage*-direktiivi. On suositeltavaa, että jokaista jatkuvan toimituksen erillistä osaa kohden luodaan oma vaihe. Esimerkiksi kääntäminen, testaaminen ja tuottaminen pitäisi tapahtua erillisillä vaiheillaan [27]. Vaiheiden avulla monet liitännäiset pystyvät esittämään visuaalisesti liukuhihnan tilaa ja edistymistä [26].

Liukuhihnassa voidaan määritellä *parallel*-lohko, jonka sisällä olevat vaiheet suoritetaan rinnakkain. Yleisesti vaihe voi sisältää vain yhden *steps*-, *stages*- tai *parallel*-lohkon. Rinnakkain suoritettavat vaiheet eivät voi itsessään sisältää rinnakkaisia vaiheita, mutta muutoin ne toimivat kuten muutkin vaiheet. Lisäksi lisäämällä komento *failFast true* lohkon *parallel* sisältämään vaiheeseen on mahdollista keskeyttää muiden rinnakkaisten vaiheiden suoritus välittömästi, mikäli yksikin niistä epäonnistuu. [27]

Steps-lohko sisältää vähintään yhden askeleen (step), jotka suoritetaan järjestyksessä [27]. Askeleet ovat käytännössä joko suoria komentoja tai funktiokutsuja. Esimerkiksi muita liukuhihnoja voidaan suorittaa komennolla *build job: 'liukuhihnanNimi'* ja niille voi tarvittaessa syöttää parametreja.

3.2 Travis CI

Travis CI on pääasiassa jatkuvaan integraatioon tarkoitettu ilmainen ohjelmisto, mutta se soveltuu myös jatkuvaan toimitukseen. Sitä mainostetaan etenkin GitHub-projektien testaamiseen soveltuvana ohjelmistona. Travis CI tarjoaa monia tietokantoja ja palveluita esiasennettuina, joita voi ottaa käyttöön kääntämiskonfiguraatiossa. Jokainen koodiintegrointipyyntö (pull request) pystytään testaamaan ennen varsinaista integrointia. Tuot-

taminen voidaan suorittaa aina automaattisesti, kun laaditut testit on suoritettu onnistuneesti. [45]

Ajettaessa käännöstä Travis CI kloonaa GitHub-repositorion uuteen virtuaaliympäristöön, ja suorittaa sarjan tehtäviä kääntääkseen ja testatakseen koodia. Jos yksikin tehtävä epäonnistuu, käännös on rikkinäinen. Muussa tapauksessa käännös on onnistunut, ja Travis CI voi tuottaa koodin web-palvelimelle tai sovelluksen verkkolevylle. Käännökset voivat myös automatisoida toimituksen muita osia, joten työtehtävät voivat olla riippuvaisia toisistaan. [44]

3.3 CircleCI

CircleCI on sekä jatkuvaan integraatioon että toimitukseen perustuva ohjelmisto, joka on peruskäytöltään ilmainen. Sen avainsanoja ovat tehokkuus, joustavuus ja hallittavuus, ja sen luvataan nopeuttavan ohjelmistokehitystä ja tarjoavan paljon muokattavuutta. Se automatisoi ohjelmistokehityksen liukuhihnan toimintaa koodin versionhallintaan viemisistä ohjelmiston julkaisuun asti. CircleCI on integroitavissa lähdekoodialustojen, kuten GitHubin, GitHub Enterprisen ja Bitbucketin, kanssa. Jokaisen versionhallintaan viemisen yhteydessä CircleCI luo käännöksen, jota testataan joko kontissa tai virtuaalikoneella. Käännöksen epäonnistuessa siitä luodaan ilmoitus, jotta ongelmat ovat nopeasti kehittäjien nähtävillä ja korjattavissa. Käännökset, jotka läpäisevät testit, ovat tuotettavissa moniin eri ympäristöihin, jolloin tuote saadaan kaupalliseen käyttöön nopeasti. [8]

CircleCI sisältää ominaisuuksia testauksen automatisoinnille sekä virtuaalikoneilla että konteissa. Käännösten epäonnistumisista ilmoitetaan välittömästi, jolloin virheet voidaan korjata aikaisessa vaiheessa. CircleCI:ssä on liukuhihnoja vastaavia työvoita (workflow), joilla voidaan määrittää käännösten, testauksen ja tuottamisen kulku. Docker-tuen ansiosta käyttäjät voivat konfiguroida ympäristönsä juuri haluamakseen. Mikä tahansa Docker-kuva on ajettavissa Dockerin rekistereistä, ja ajamista voidaan muokata tehtäväkohtaisesti. [9]

Käyttäjät voivat konfiguroida resussinsa perustuen tiiminsä käännösten tarpeille. CircleCI tukee myös kaikkia ohjelmointikieliä, joita voidaan kääntää Linuxilla tai macOS:lla, kuten C++, Javascript, PHP, Python ja Ruby. Kehittäjät voivat hyödyntää haluamiaan työkaluja ja viitekehyksiä, ja käännöksiä voidaan nopeuttaa CircleCI:n välimuistiasetusten avulla. Välimuistin käyttö on konfiguroitavissa Docker-kuville, lähdekoodille ja riippuvuuksille. CircleCI:n tietoturvateknologioihin sisältyy muun muassa kokonaisvaltainen virtuaalikoneiden eriytys, Dockerin tietoturvakyvyt sekä Linuxin käyttöjärjestelmäytimen tietoturvamoduulit. Kehittäjät voivat tarkastella käännöksiä CircleCI:n interaktiivisella käyttöliittymällä. Käyttöliittymä on muokattavissa siten, että siitä saa näkyviin esimerkiksi useimmiten epäonnistuneet tai hitaimmat testit. [9]

3.4 Codeship

Codeship on peruskäytöltään ilmainen CI/CD-ohjelmisto, jota tarjotaan pilvipalveluna ja joka nopeuttaa ohjelmistokehitystä. Se tarjoaa muokattavissa olevaa automaatiota, jonka avulla ohjelmakoodimuutoksia saadaan vietyä tuotantoon nopeammin. Kääntämis- ja testausautomaatiota hyödyntämällä voidaan löytää virheitä koodissa ennen tuotantoon julkaisemista, jolloin sovelluksen toimivuus tuotantoympäristössä saadaan varmistettua. Codeship tarjoaa rinnakkaisten liukuhihnojen hyödyntämisen, samanaikaisen kääntämisen ja välimuistin hyödyntämisen, mikä tekee siitä tehokkaan työkalun. [10]

Codeshipin vahvuuksia ovat nopeus, luotettavuus ja yksinkertaisuus. Sen voi konfiguroida kääntämään ja tuottamaan GitHubissa olevan sovelluksen demo- tai tuotantoympäristöön. Se tarjoaa monia asetuksia ympäristön alustamiseen. CodeShip on integroitu suosituihin lähdekoodialustoihin GitHubiin ja Bitbuckettiin, ja se tukee monia ohjelmointikieliä ja tuotantoalustoja. Kun koodia viedään GitHubiin tai Bitbuckettiin, CodeShip kääntää sovelluksen turvallisilla palvelimillaan ja ajaa laaditut automatisoidut testit. Jos testit epäonnistuvat, se huomauttaa kehitystiimiä sähköpostitse tai jonkin integroidun palvelun kautta, kuten Slackissa tai HipChatissa. Jos testit onnistuvat, CodeShip vie käännöksen demo- tai tuotantopalvelimille määritellyn liukuhihnan mukaisesti. [35]

4. JATKUVA TOIMITUS ERASSA

Työn kohdejärjestelmäksi valittiin Atostek Oy:n kehittämä web-palvelu eRA, jonka avulla sosiaali- ja terveydenhuollon tietojärjestelmiä voidaan liittää Kelan tarjoamiin Kansallisen terveysarkiston palveluihin eli Kanta-palveluihin [3]. Sen avulla terveydenhuollon toimijat voivat ottaa Kanta-palveluita helposti ja nopeasti käyttöön. Palvelu tarjoaa valmiit käyttöliittymät eri palveluiden käyttöön, kuten sähköiseeen lääkemääräykseen, potilastiedon arkistoon sekä lomakepalveluun [4]. Kuvassa 4.1 on havainnollistettu tietojärjestelmän liittäminen Kanta-palveluihin.

Palvelun eRA avulla sovelluksia ja tietojärjestelmiä voidaan integroida Kanta-palveluihin vain yhden rajapinnan kautta ilman raskasta yhteistestausta ja sertifiointia. Tietojärjestelmiä voidaan integroida käyttämään Kelan sähköistä lääkemääräystä web-käyttöliittymän kautta. Potilastietojärjestelmiä voidaan liittää Kelan potilastiedon arkistoon web-käyttöliittymässä tai ohjelmointirajapintojen kautta. Lomakepalvelun avulla Kannasta voidaan hakea ja tallentaa sekä lääkärintodistuksia että lausuntoja. Lisäksi vanhojen tietojen katselimella voidaan tarkastella vanhoja potilastietoja, joita on siirretty Kantaan aiemmasta potilastietojärjestelmästä. [5]

4.1 eRAn kehittäminen

Palvelun eRA kehittämisessä käytetään pääasiassa Visual Studiota, joka on integroitu ohjelmointiympäristö (IDE, Integrated Development Environment). Sen ominaisuuksien ansiosta koodin kirjoittaminen, virheiden etsiminen ja korjaaminen sekä testaaminen onnistuvat kätevästi. Se on myös laajasti muokattavissa ja yhdistettävissä versionhallintajärjestelmiin [47]. Palvelun eRA ja siihen liityvien simulaattoreiden ohjelmoimisessa käytetään C#-ohjelmointikieltä, joka on Microsoftin kehittämä oliopohjainen ohjelmointikieli. Olio-ohjelmoinnissa pyritään tiedon modulaarisuuteen hyödyntämällä olioita, joihin sidotaan vain niiden itsensä tarvitsemaa tietoa ja toiminnallisuutta.

Kuva 4.1. *Tietojärjestelmän liittäminen Kanta-palveluihin eRAn välityksellä [5].*

Kehityksessä hyödynnetään ASP.NET:iä, joka on avoimeen lähdekoodiin perustuva viitekehys nykyaikaisten web-sovellusten ja -palveluiden rakentamiselle. Sen avulla voidaan luoda yksinkertaisia, nopeita ja skaalautuvia web-sivustoja [2]. Tarkemmin ottaen kehitys tapahtuu ASP.NET MVC-laajennoksella, joka perustuu MVC-mallin (model-view-controller, suom. malli–näkymä–käsittelijä) hyödyntämiselle. MVC-mallissa käyttäjä tarkastelee tietoa näkymässä, ja käsittelijä on vuorovaikuksessa käyttäjän toimenpiteiden kanssa. Näkymä ja käsittelijä muodostavat keskenään sovelluksen käyttöliittymän. Tietomalli on sovelluksen osa, joka sisältää sekä näkymän että logiikan tarvitsemaa tietoa, ja muokkaa kyseisiä tietoja käyttäjän vuorovaikutuksen seurauksena. [31]

Seuraavia simulaattoriprojekteja kehitetään tiiviisti eRAn yhteydessä:

- SimArkisto
- SimReseptikeskus
- SimValvira.

SimArkisto, SimReseptikeskus ja SimValvira ovat yrityksellä kehitettyjä simulaattoreita, joita kehitys- ja demoympäristössä sijaitseva eRA hyödyntää eri tilanteissa lähettämällä niille pyyntöjä ja vastaanottamalla vastauksia HTTPS-rajapinnan yli (Hypertext Transfer Protocol Secure). HTTPS on HTTP:n laajennos, jossa kommunikaatio salataan salausprotokollaa hyödyntämällä. Palvelussa eRA käytetään varmenteita (certificate), joiden avulla tunnistetaan HTTPS-rajapintaa käyttävät osapuolet. SimArkisto simuloi Kelan Kanta-arkistoa, ja sitä käytetään enimmäkseen sekä palvelutapahtumien että diagnoosien käsittelyyn. SimReseptikeskuksen kautta voidaan käsitellä erilaisia lääkemääräyksiin ja uusimispyyntöihin liittyviä asioita. SimValvira simuloi sosiaali- ja terveysalan lupa- ja valvontavirasto Valviran palveluita. SimValviran välityksellä voidaan hallinnoida ammattioikeuspalveluita sekä asettaa käyttöoikeuksia eri rooleille, kuten lääkäreille ja sairaanhoitajille.

Palvelun eRA kehittämisessä on sovellettu perinteistä toimitusprosessia jo useamman vuoden ajan, eli palvelun julkaisemisessa tuotantoon asti ei ole sovellettu automatiikkaa. Käytännössä julkaistaakseen palvelun tuotannossa työntekijän on täytynyt päivittää tarvittavien projektien koodit versionhallinnasta, kääntää projektit ja testata järjestelmää hyödyntäen sekä manuaalista regressiotestausta että tutkivaa testausta. Kun ohjelmistoa on saatu testattua riittävän hyvin, se on julkaistu tuotantoon. Ohjelmiston kasvaessa ajan myötä julkaisuihin on kulunut yhä enemmän aikaa. Täten projektissa nousi esiin tarve ketterämmälle toimitusprosessille, jossa mahdollisimman moni vaihe kehittämisen ja julkaisun välillä saataisiin automatisoitua. Prosessiksi valittiin jatkuva toimitus, koska sillä saadaan automatisoitua kaikki julkaisua edeltävät toimenpiteet.

4.2 Työkaluratkaisu

Jatkuvan toimituksen työkaluksi valittiin Jenkins. Lähtökohtaisesti Jenkinsin valinnalle ei ollut muita perusteita kuin sen suosio ja hyvä maine automaatiopalveluna. Vertailtaessa

Jenkinsiä muiden CI/CD-palvelujen kesken ilmeni, että Jenkinsille löytyy huomattavasti eniten dokumentaatiota verkosta. Lisäksi Jenkinsin lukuisista liitännäisistä löytyy tuki versionhallintajärjestelmä SVN:lle (Subversion), jota käytetään eRAssa ja siihen liittyvissä projekteissa. Monet muut CI/CD-ohjelmistot soveltuvat käytettäviksi enimmäkseen Git-versionhallintajärjestelmän kanssa. Jenkinsin suosion seurauksena sen elinkaari tulee todennäköisesti olemaan pisin muihin työkaluihin verrattuna, jolloin Jenkinsiä voidaan hyödyntää pitkään eRAssa. Lisäksi liitännäispohjaisen arkkitehtuurin ansiosta tarvittavia liitännäisiä voidaan asentaa tulevaisuuden tarpeiden mukaan.

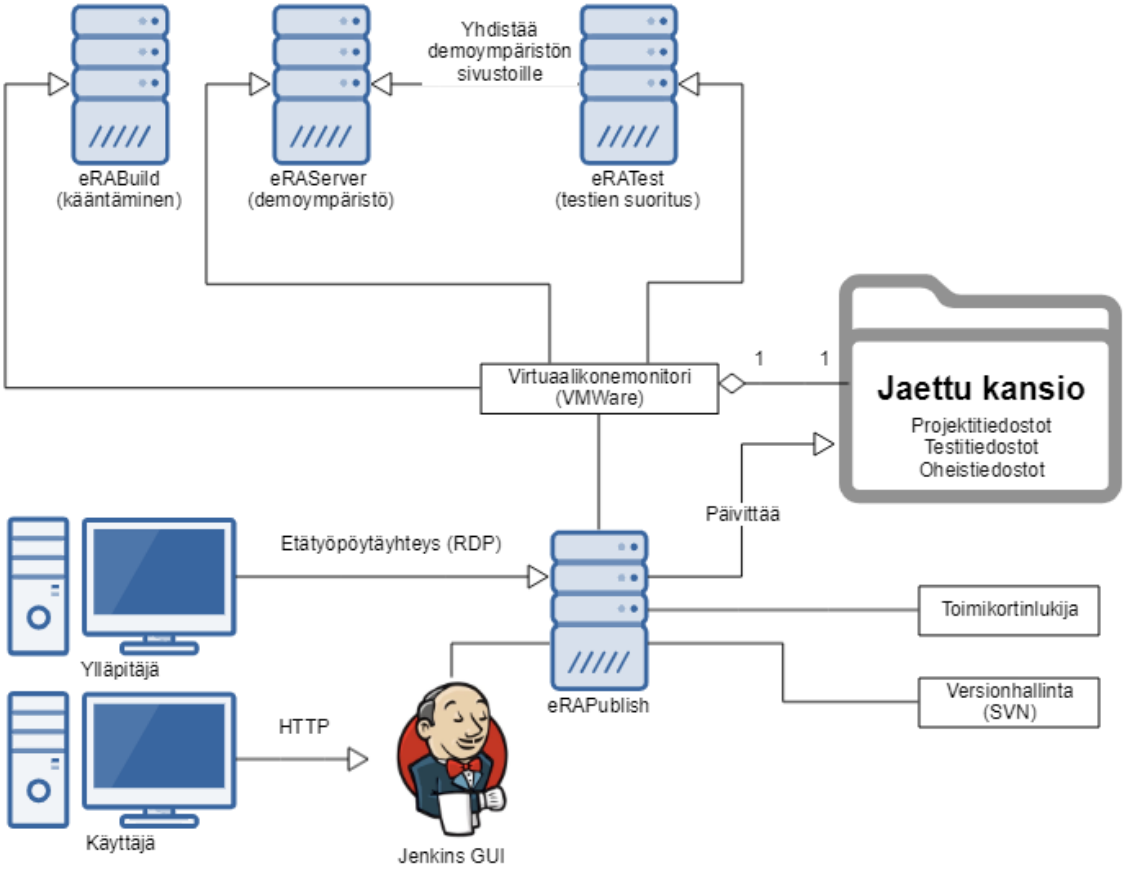
Aluksi pohdittiin myös oman työkalun tekemistä, mutta valmiisiin työkaluihin nähden sellaisen tekeminen ja ylläpitäminen olisi vienyt liikaa resursseja. Lisäksi paikallisuudella ei olisi ollut mitään etua, etenkään kun valmiissa työkaluissa tietoturva on lähes aina riittävän hyvä.

4.3 Ympäristö

Ympäristö suunniteltiin kuvan 4.2 mukaiseksi. Ympäristöksi valittiin palvelimina toimivia virtuaalikoneita hyödyntävä kokonaisuus, jonka keskeisessä osassa on ohjelmisto nimeltään VMWare Workstation Pro. Se on virtuaalikonemonitori, jonka sisällä voi luoda useita eri virtuaalikoneita, joissa voi käyttää mitä käyttöjärjestelmiä tahansa [52]. Virtuaalikoneet ovat eristyksissä olevia ympäristöjä erilaisilla yksityisyysasetuksilla, työkaluilla ja verkon konfiguraatioilla. Kaikki ympäristön käyttöjärjestelmät ovat Windows-pohjaisia eRAn vaatimusten ja yhdenmukaisuuden takia.

Yritys tarjosi suorituskykyisen palvelimen jatkuvaa toimitusta varten, jonka päälle rakennettiin virtuaalikoneympäristö. Toinen vaihtoehto olisi ollut hyödyntää kontitusta, jolloin virtuaalikoneympäristöä ei olisi tarvittu ja ainut tarvittava käyttöjärjestelmä olisi jäänyt palvelimelle. Kontitukseen tarkoitetun Dockerin käytöstä ei projektin sisällä ollut juurikaan kokemusta, joten sen hyödyntämistä tutkittiin parin viikon ajan. Dockerin konfigurointi koitui haastavaksi Windows-ympäristössä, eikä MSBuild- tai MSDeploy-työkaluja saatu toimimaan konteissa. Täten Dockerin hyödyntämisestä päätettiin luopua ja keskityttiin alkuperäiseen suunnitelmaan virtuaalikoneympäristöstä. Vaikka virtualisointi tuhlaakin resursseja, palvelin on tarpeeksi tehokas pyörittääkseen virtuaalikoneympäristöä vaivattomasti. Lisäksi tuotantoympäristössä käytetään samaa käyttöjärjestelmää ja samoja ohjelmistoja kuin demoympäristö eRAServerillä, joten virtualisoinnin avulla tuotantoympäristöä voidaan matkia erittäin tarkasti, kun taas kontituksella matkiminen ei olisi yhtä selkeää.

Ympäristö koostuu neljästä palvelimesta, jotka ovat pääpalvelimena toimiva eRAPublish sekä virtuaalikoneet eRABuild, eRAServer ja eRATest. eRAPublish ja virtuaalikoneet sijaitsevat omassa sisäverkossaan, jonka sisällä ne kommunikoivat keskenään tarvittaessa. Esimerkiksi eRATest-koneella suoritettavat testit hyödyntävät eRAServer-konetta, joka puolestaan pyörittää eRAn ja simulaattoreiden web-sivustoja. eRAPublish toimii myös



Kuva 4.2. *Jatkuvan toimituksen ympäristö eRAssa.*

ulkoverkossa, jotta siihen voidaan ottaa etätyöpöytäyhteys. Lisäksi eRAServer pääsee ulkoverkkoon, koska osassa siellä suoritettavista käyttöliittymätesteistä täytyy ladata kolmansien osapuolien varmenteita ulkoverkosta. Taulukossa 4.1 on esitetty ympäristön koneiden IPv4-osoitteet (Internet Protocol) sisä- ja ulkoverkossa.

Taulukko 4.1. *Palvelinten IP-osoitteet.*

Tietokone	Sisäverkon IPv4 osoite	Ulkoverkon IPv4-osoite
eRAPublish	192.168.2.1	10.80.50.102
eRABuild	192.168.2.11	Ei ole
eRAServer	192.168.2.12	10.80.50.109
eRATest	192.168.2.10	Ei ole

Palvelimille on asennettu ohjelmia, joita tarvitaan tehtävien suorittamisen aikana. Asennetut ohjelmat on koottu taulukkoon 4.2. Asennettujen ohjelmien määrä on pyritty minimoimaan tehokkuuden ja yksinkertaisuuden vuoksi. Virtuaalikone eRAServeriä lukuunottamatta jokaisella koneella on käyttöjärjestelmänä Windows 10. Tuotantoympäristössä käytetään käyttöjärjestelmänä Windows Server 2016:ta, joten sen peilaamisen vuoksi demoympäristönä toimiva eRAServer käyttää samaa käyttöjärjestelmää. Jokaiselle koneelle on asennettu selaimeksi Google Chrome, jolla voi vähintäänkin ohjata Jenkinsin käyttöliittymää, vaikkei kone olisikaan yhdistettynä ulkoverkkoon.

Taulukko 4.2. *Palvelimille asennetut ohjelmat.*

Palvelin	Ohjelman nimi
Kaikki	Google Chrome
eRAPublish	Jenkins VMWare Workstation Pro SVN
eRABuild	MSBuild 15.0
eRAServer	IIS SQL Server Management Studio 2016 Microsoft Web Deploy 3.6
eRATest	eRASmartCard Robot Framework 3.0.2 InterfaceTesterTool

4.3.1 eRAPublish

eRAPublish on palvelimena toimiva fyysinen tietokone. Sillä on kolme päätehtävää:

- Jenkinsin pyörittäminen
- virtuaalikoneiden pyörittäminen
- eRAan liittyvien projektien tiedostojen päivittäminen ja jakaminen sisäverkossa.

eRAPublishin tärkein tehtävä on Jenkinsin pyörittäminen. Jenkinsin asennusohjelma ladataan sen web-sivujen kautta, jonka avulla Jenkins saadaan asennettua tietokoneelle. Jenkins pyörii eRAPublishilla *localhostina* eli se käyttää samaa IP-osoitetta kuin eRAPublish, mutta Jenkinsille on varattu oma porttinumero. Täten Jenkinsiin pääsee käsiksi kaikilta yrityksen verkossa olevilta koneilta. Vahinkojen ja väärinkäyttöjen estämiseksi ainoastaan sisäänkirjautuneet käyttäjät voivat suorittaa tehtäviä manuaalisesti ja muokata konfiguraatiota.

Virtuaalikoneet pyörivät eRAPublishin sisältämällä virtuaalikonemonitori VMWarella, jossa ne suorittavat vain niille ominaisia tehtäviä. Niille on annettu tarpeen mukaan resursseja, kuten suoritettavien tehtävien edellyttämiä ohjelmistoja. Jokainen virtuaalikone on asetettu pysymään päällä koko ajan, joten ne ovat aina käytettävissä.

Projekteihin liittyy tiedostoja, joiden jakaminen on tarpeellista eRAPublishin ja virtuaalikoneiden kesken. VMWaren avulla on luotu jaettu hakemisto eRAPublish-koneelle, johon virtuaalikoneet pääsevät käsiksi. Jaettu hakemisto sisältää hakemistot jokaiselle projektille, demoympäristön alustuksessa tarvittaville tiedostoille, käyttöliittymätestien projektille, rajapintatestaustyökalulle sekä käyttöliittymätestien että rajapintatestien tuloksille. Tiedostot ovat peräisin SVN:n eri säilytyspaikoista (repository), joista tiedostot päivitetään jaettuun hakemistoon liukuhihnan suorituksen alussa.

eRAPublishiin voidaan ottaa yhteys RDP:llä (Remote Desktop Protocol, suom. etätyöpöytäyhteys). RDP tarjoaa etäisen näytön ja ohjauksen Windows-käyttöjärjestelmien kesken, jotka sijaitsevat samassa verkossa. RDP on suunniteltu tukemaan erilaisia verkkotopologioita ja lähiverkon protokollia [36]. Vain ylläpitäjien täytyy pystyä yhdistämään

eRAPublishiin RDP:llä, joten yhdistäminen edellyttää autentikointia. Koska eRAPublishilla pyörivää Jenkinsiä pystyy käyttämään HTTP-rajapinnan kautta, RDP:n käyttäminen on tarpeellista vain silloin, kun itse eRAPublishilla tai sen virtuaalikoneilla täytyy tehdä muutoksia.

4.3.2 eRABuild

eRABuild on virtuaalikone, jonka tehtävä on toimia kääntämisympäristönä eri projekteille. Se toimii vain sisäverkossa, koska sillä ei ole koskaan tarvetta päästä ulkoverkkoon.

eRABuildille on asennettu kääntämistyökalu MSBuild 15.0. Kehitysvaiheessa eRAn projekteissa käytetään integroitua ohjelmointiympäristöä nimeltään Visual Studio, joka itsessään hyödyntää MSBuildia projektia käännettäessä. Ympäristössä ei ole kuitenkaan tarvetta Visual Studiolle asentamiselle, koska se veisi turhaan muistia ja kääntäminen onnistuu suoraan MSBuildilla, joka ei ole riippuvainen Visual Studiosta. Siten MSBuildilla voidaan kääntää sekä *project*- että *solution*-tyyppisiä tiedostoja ympäristössä, jossa Visual Studio ei ole asennettuna [33]. Projektitiedostot ovat XML-pohjaisia (Extensible Markup Language, rakenteellinen kuvauskieli) tiedostoja, joiden avulla voidaan määrittää eri asetuksia kääntämiselle. MSBuildiä pystyy ajamaan suoraan komentoriviltä, jolloin esimerkiksi käännöksen tiedot päivittyvät Jenkinsin lokeihin kätevästi. Lisäksi komentoriviltä ajettaessa MSBuildille pystyy syöttämään kääntämiseen vaikuttavia parametreja.

4.3.3 eRAServer

eRAServer on virtuaalikone, joka toimii demoympäristönä. Sen tarkoituksena on matkia tuotantoympäristöä mahdollisimman tarkasti, jotta voitaisiin varmistaa palvelun toimivuus ja arvioida, miten palvelu ilmenee loppukäyttäjillä. Täten eRAServerille on asennettu sama käyttöjärjestelmä ja samoja ohjelmistoja kuin mitä tuotantoympäristössä käytetään. eRAServerillä on yhteys ulkoverkkoon, koska osa eRATest-koneelta ajettavista testeistä vaatii HTTP-pyyntöjen lähettämistä ulkoverkkoon ja viestien vastaanottamista ulkoverkosta.

eRAServerille on asennettu Microsoft Web Deploy 3.6, jolla voidaan tuottaa web-sovelluksia ja -sivustoja IIS-palvelimille [48]. Visual Studio käyttää Web Deployta, mutta Web Deployta pystyy käyttämään myös suoraan komentorivin kautta. Kuten MSBuildin tapauksessa, Visual Studiota ei ole tarpeellista asentaa virtuaalikoneelle vastaavista syistä.

eRAServerille on asennettu IIS (Internet Information Services), joka on Windows Server -käyttöjärjestelmille tarkoitettu joustava, turvallinen ja helposti käsiteltävä palvelinohjelmisto, jolla voidaan tarjota palveluita verkossa. Se pystyy skaalautuvan ja avoimen arkkitehtuurinsa ansiosta käsittelemään monia tehtäviä, kuten median suoratoistoa sekä web-sovellusten ylläpitoa [24]. IIS:ään pystyy lisäämään sivustoja, jolloin niille asetetaan nimet, fyysinen polku (physical path), IP-osoite sekä porttinumero. Fyysinen polku tarkoittaa hakemistoa, jonne tuottamisvaiheessa luodaan tarvittavat tiedostot. eRAlle ja sen

simulaattoreille luotiin sivut omilla porttinumeroillaan ja fyysisilla poluillaan. Sivujen IP-osoitteiksi asetettiin *localhost*, eli ne pyörivät paikallisesti eRAServer-koneella.

Lisäksi eRAServerille on asennettu tietokantaohjelmisto SQL Server Management Studio 2016. Se on integroitu ympäristö, jossa voidaan käsitellä, konfiguroida, hallita ja kehittää eri SQL-infrastruktuureja (Structured Query Language). Se tarjoaa kattavan työkalun, jossa on lukuisia graafisia työkaluja skriptieditoreineen. Siten SQL Server soveltuu kaikentaitoisille kehittäjille sekä tietokantojen hallinnoijille [38]. Demoympäristön alustuksen yhteydessä jokaiselle projektille luodaan tietokannat, jonne alustetaan tarvittavat tietokantataulut. Projektin eRA tietokanta sisältää kymmeniä tauluja, joihin muun muassa sisältyvät taulu palvelutapahtumille, rekisteröidyille käyttäjille ja koodistoille. Projektit ovat yhteydessä niille ominaisiin tietokantoihin, ja eri toimenpiteiden yhteydessä tietokantoihin lisätään tietoa. Esimerkiksi eRAssa uuden istunnon aloittamisen seurauksena *SystemSessions*-tauluun lisätään rivi, joka sisältää tietoa kyseisestä istunnosta. Projekteissa hyödynnetään tietokannassa olevaa tietoa kooditasolla. Muun muassa eRAan kirjautuessa tutkitaan, onko kirjautuneelle käyttäjälle olemassa keskeneräistä istuntoa.

4.3.4 eRATest

eRATest on virtuaalikone, jolla suoritetaan sekä käyttöliittymä- että rajapintatestausta. Muista virtuaalikoneista poiketen Jenkins-agentti käynnistetään komentorivin kautta eRATest-koneella. Se on välttämätöntä, koska käyttöliittymätestit vaativat selaimen ja eRASmartCard-ohjelmiston ikkunoiden olevan manipuloitavissa eRATestin käyttöliittymän kautta. Agentin käynnistäminen onnistuu suorittamalla alla oleva batch-komento.

```
java -jar C:\Jenkins\slave.jar -jnlpUrl http://eRAPublish:8080/computer
    /eRATest/slave-agent.jnlp
```

eRATest-koneelle on asennettu yrityksellä kehitetty eRASmartCard-toimikortinlukijaohjelmisto, jota käytetään eRA-palvelussa sekä sisäänkirjautumiseen että sähköiseen allekirjoitukseen. Testejä varten luotu testikortti on yhdistettynä toimikortinlukijaan, joka puolestaan on yhdistettynä fyysiseen eRAPublish-palvelimeen. Virtuaalikonemonitori puolestaan tarjoaa ajurin kortinlukijalle, jonka avulla lukija on käytettävissä eRATest-koneella. Toimikorttia on välttämätöntä käyttää, sillä käyttöliittymätestit sisältävät lukuisia kohtia, joissa eRA vaatii joko sisäänkirjautumista tai sähköistä allekirjoitusta.

Kohdejärjestelmälle suoritetaan sekä käyttöliittymä- että rajapintatestausta liukuhihnan viimeisissä vaiheissa. Molemmat testaukset suoritetaan demoympäristössä sen jälkeen, kun ohjelmisto on tuotettu kyseiseen ympäristöön, tietokannat on nollattu ja ympäristö on alustettu toimintakuntoon. Testausten lopputuloksina syntyvät raportit, joiden pohjalta tehdään päätös, julkaistaanko ohjelmisto tuotantoon loppukäyttäjille.

4.4 Liukuhihnan suoritus

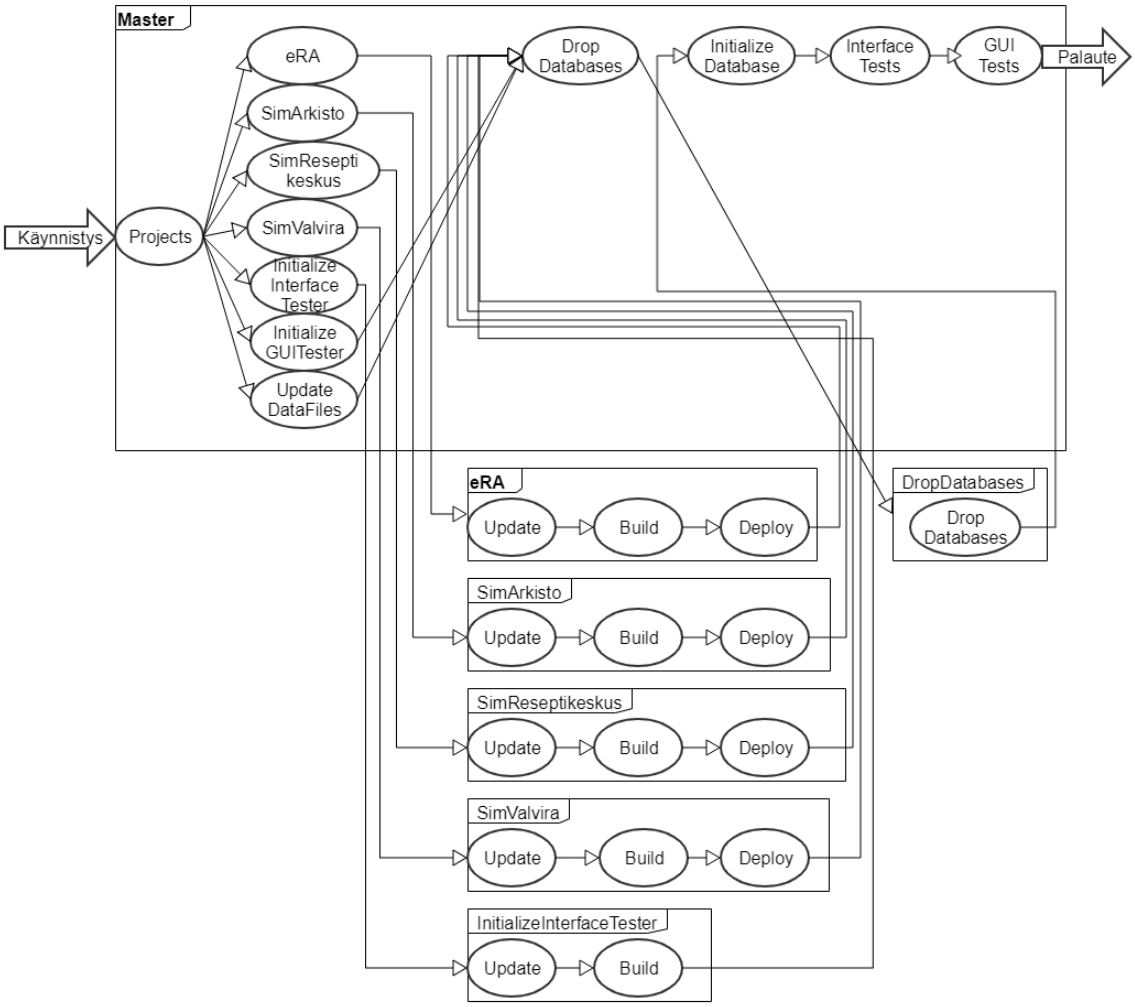
Jatkuva toimitus eRAssa perustuu Jenkinsillä luodun Master-nimisen liukuhihnan säännölliseen suorittamiseen pääpalvelimella. Liukuhihna ei itsessään suorita jokaista vaihetta, vaan se kutsuu lisäksi muita tarvittavia tehtäviä ja liukuhihnoja. Master-liukuhihna suoritetaan jokaisen päivän vaihtuessa, ja sitä tai sen sisältämiä liukuhihnoja voidaan suorittaa tarpeen mukaan manuaalisesti. Liukuhihna tuottaa tuloksia sen vaiheista pitkin suoritusta, ja sen päättyessä saadaan lopputulos liukuhihnan kokonaisvaltaisesta onnistumisesta. Liukuhihna sisältää niin sekventiaalista kuin rinnakkaista tehtävien suoritusta. Taulukossa 4.3 on esitetty liukuhihnat ja tehtävät, joita suoritetaan jatkuvan toimituksen aikana.

Taulukko 4.3. *Liukuhihnat ja tehtävät, joita suoritetaan eRAn toimittamisen aikana.*

Nimi	Lyhyt kuvaus	Kategoria
Master	Suorittaa jatkuvalta toimitukselta edellytetyt toimenpiteet	Liukuhihna
eRA	Päivittää, kääntää ja tuottaa eRAn	
SimArkisto	Päivittää, kääntää ja tuottaa SimArkiston	
SimReseptikeskus	Päivittää, kääntää ja tuottaa SimReseptikeskuksen	
SimValvira	Päivittää, kääntää ja tuottaa SimValviran	
InitializeInterfaceTester	Päivittää ja kääntää rajapintatestaussovelluksen	
DropDatabases	Tyhjentää ympäristön tietokannat	
Build	Kääntää projektin	Tehtävä
Deploy	Tuottaa projektin demoympäristöön	
DropDatabase	Tyhjentää tietokannan	
GUITest	Suorittaa käyttöliittymätestit	
InitializeDatabase	Alustaa tietokannat ja demompäristön	
InterfaceTest	Suorittaa rajapintatestit	
ResetIIS	Käynnistää IIS:n uudelleen	
Update	Päivittää projektin tiedostot	

Kuvassa 4.3 on esitetty liukuhihnojen suoritus jatkuvassa toimituksessa. Siinä suorakaiteet edustavat liukuhihnoja ja ellipsit liukuhihnojen sisältämiä vaiheita. Vaiheiden sisältämistä askeleista aloitetaan muiden liukuhihnojen ja tehtävien suoritus. Liukuhihnalla suoritetaan ensin projektikohtaiset liukuhihnat rinnakkain. Projektikohtaisilla liukuhihnoilla suoritetaan kunkin projektin päivitys, kääntäminen ja demoympäristöön tuottaminen. Rajapintatestaussovelluksen alustamisessa riittää pelkästään projektin päivitys ja kääntäminen, koska se ei ole tuotettava web-palvelu. Käyttöliittymätestien ja demoympäristön alustuksessa tarvittavat tiedostot päivitetään, mutta niiden yhteydessä ei ole mitään käännettävää eikä tuotettavaa ohjelmaa. Jos jokainen rinnakkaisista vaiheista saadaan suoritettua onnistuneesti, liukuhihnalla edetään tietokantojen tyhjentämiseen ja sen jälkeen demoympäristö alustetaan toimintakuntoon. Viimeiseksi suoritetaan rajapinta- ja käyttöliittymätestausta.

Liukuhihnojen arkkitehtuuri koki muutoksia toteuttamisen aikana. Aluksi luotiin vain yksi pitkä liukuhihna, jonka tarkoitus oli suorittaa kaikki jatkuvassa toimituksessa vaaditut toimenpiteet. Koodin määrän kasvaessa sekä ylläpidettävyyden hankaloituessa huomattiin, että koodia on järkevä hajauttaa projektikohtaisesti, joten jokaiselle projektille



Kuva 4.3. *Liukuhihnojen suoritus eRAn toimituksessa.*

luotiin oma liukuhihna. Silloin kuitenkin havaittiin, että monissa liukuhihnoissa suoritettiin samankaltaisia toimenpiteitä, kuten tiedostojen päivittämistä ja ohjelman kääntämistä. Täten kyseiset toimenpiteet vietiin erillisiksi tehtäviksi, joita kutsutaan projektien liukuhihnoilla projektikohtaisia parametreja käyttäen. Liitteissä A ja B on esitetty sekä Master- että eRA-liukuhihnojen toteutukset. Muiden liukuhihnojen toteutuksia ei esitetä, koska ne ovat hyvin samankaltaisia ja turhan yksityiskohtaista tietoa työn kannalta.

Batch-komennoissa esiintyvät muuttujat ja niiden merkitykset on esitetty taulukossa 4.4. Muuttujat edustavat tarvittavien tiedostojen polkuja ja hakemistoja, joita käsitellään batch-komennoissa. Muuttujille asetetaan arvot batch-komennoissa ennen varsinaisen tehtävän suorittamista. Luonnollisesti on mahdollista kirjoittaa muuttujien arvot suoraan komentoihin, mutta muuttujien avulla komentoja on helpompi lukea. Itse muuttujien arvoja ei ole esitetty, koska ne olisivat turhan yksityiskohtaista tietoa työn kannalta.

***Taulukko 4.4**. Muuttujat batch-komennoissa.*

Komento	Muuttuja	Merkitys
Update	SVN	Svn-ohjelman polku
Build	MSBUILD	MSBuild-ohjelman polku
Deploy	DEPLOY_PATH	Microsoft Web Deploy -ohjelman polku
InterfaceTest	TOOL_PATH	Komentorivityökalun polku
	TESTCHAIN_PATH	Testiketjujen polku
	CONFIG_PATH	Konfiguraatiotiedoston polku
	ERROR_DATABASE_PATH	Paikallisen virhetietokannan polku
	OUTPUT_PATH	Testaustulosten polku

4.4.1 Päivittäminen, kääntäminen ja tuottaminen demoympäristöön

Update on eRAPublish-koneella suoritettava tehtävä, joka päivittää tietyssä hakemistossa sijaitsevat tiedostot. Se ottaa parametrina päivitettävän polun (*updatePath*) ja päivittää sen sisältämät tiedostot alla esitetyn komennon mukaisesti.

```
"%SVN%" update "%updatePath%" --non-interactive --no-auth-cache --
   username erabuilder --password ********
```

Koska projektien tiedostot sijaitsevat versionhallintajärjestelmä SVN:ssä, kohdehakemistolle suoritetaan komento *svn update*. Päivittäminen vaatii autentikointia, joten komennon parametreina syötetään yleiseen tarkoitukseen suunnitellut käyttäjätunnus ja salasana. Komentoa voidaan kutsua mille tahansa polulle, mutta työn kontekstissa jokainen päivitettävä polku sijaitsee virtuaalikonemonitorin kautta jaetussa kansiossa. Jos päivitys epäonnistuu esimerkiksi koodikonfliktien takia, komento palauttaa virhekoodin ja liukuhihnan suoritus keskeytyy. Päivittämisen onnistuessa voidaan jatkaa kääntämiseen.

Build on eRABuild-virtuaalikoneella suoritettava tehtävä, joka kääntää projektin hyödyntämällä MSBuild-käännöstyökalua ja julkaisuprofiilia (publish profile). Tehtävä ottaa parametreinaan projektitiedoston hakemiston (*projectFilePath*), projektitiedoston nimen ilman tiedostopäätettä (*projectFileName*) sekä julkaisuprofiilin nimen ilman tiedostopäätettä (*profile*) alla esitetyn komennon mukaisesti.

```
"%MSBUILD%" "%projectFilePath%\%projectFileName%.csproj" /P:
   DeployOnBuild=true /P:PublishProfile="%projectFilePath%\Properties\
   PublishProfiles\%profile%.pubxml"
```

Julkaisuprofiilitiedosto on XML-pohjainen tiedosto, jolla voidaan määrittää monia eri asetuksia käännökselle. Projektin profiilitiedostoissa määritellään muun muassa polku

käännöksen tuottamille tiedostoille, IIS-sivuston nimi, tietokanta-asetukset sekä web-konfiguraatiot. Käännös tuottaa muun muassa zip-paketin ja parametritiedoston. Paketti sisältää sivuston tuottamiseen tarvittavat tiedostot, kuten käännöksiä, tietokantaskriptejä ja resursseja. Parametritiedostossa voidaan asettaa arvoja eri asetuksille, jotka vaikuttavat web-sovelluksen tuottamiseen Microsoft Web Deployta käytettäessä [30]. Projektien parametritiedostoissa on annettu yhteysmerkkijonoja (connection string), joiden avulla sivustot saavat yhteyden tarvitsemiinsa tietokantoihin.

Deploy on eRAServer-virtuaalikoneella suoritettava tehtävä, jolla projekti saadaan tuotettua demoympäristöön. Tuottaminen onnistuu käyttämällä Microsoft Web Deploy -työkalua komentorivin kautta. Parametrina *profile* annetaan käännöksen tuottamien tiedostojen polku, jossa sijaitsevat sekä zip-paketti että parametritiedosto. Parametrina *project* puolestaan annetaan kyseisten tiedostojen nimi ilman tiedostopäätteitä. Tiedostojen nimi vastaa tuotettavan projektin nimeä, joten se on sama kummassakin tiedostossa. Alla on esitetty komento, joka suoritetaan tehtävän aikana.

```
"%DEPLOY_PATH%" -verb:sync -source:package="%profile%\%project%.zip" -
    dest:auto,includeAcls="False" -setParamFile:"%profile%\%project%.
    SetParameters.xml"
```

Komennossa tietolähteeksi määritellään profiilin zip-paketti, joka luotiin kääntämisen yhteydessä. Komentoon syötetään parametritiedoston polku, jossa sijaitsevasta parametritiedostosta luetaan tuottamiseen vaikuttavia asetuksia. Onnistuneen tuottamisen seurauksena IIS:n fyysisen polun tiedostot päivittyvät ajan tasalle, jolloin muutokset ovat suoraan nähtävissä IIS:n pyörittämillä sivustoilla.

4.4.2 Tietokantojen tyhjentäminen

DropDatabase on eRAServer-virtuaalikoneella suoritettava tehtävä, jolla voidaan tyhjentää tietty tietokanta. Se ottaa parametrina *database* tyhjennettävän tietokannan nimen alla esitetyn komennon mukaisesti.

```
sqlcmd -U sa -P ******** -S .\ERADATABASE -Q "alter database [%database
    %] set single_user with rollback immediate; drop database [%
    database%]"
```

Tehtävässä hyödynnetään tietokantatyökalua nimeltään *sqlcmd*, jolle voidaan syöttää Transact-SQL-lauseita, proseduureja ja skriptejä eri muodoissa [41]. Komentoon syötetään sekä tietokannan käyttäjätunnus että käyttäjäkohtainen salasana. Komennossa yhdistetään

SQL Serverin instanssiin nimeltä .*ERADATABASE*. Lisäksi komennossa voidaan määritellä yksi tai useampi suoritettava tietokantaproseduuri. Proseduurilla *alter database* voidaan muokata tietokannan tiettyjä konfiguraatioasetuksia. Tietokanta asetetaan tilaan, jossa vain yksi käyttäjä kerrallaan voi käyttää tietokantaa. Jos komentoa ei voida suorittaa heti, muut samanaikaiset tietokantamuutokset perutaan *rollback immediaten* ansiosta.

DropDatabases on eRAServer-virtuaalikoneella suoritettava liukuhihna. Se suorittaa DropDatabase-tehtävän jokaiselle kohdejärjestelmän tietokannalle, eli tyhjentää tietokannat eRADatabase, SimArkisto, SimReseptikeskus, SimValvira, SupportDatabase ja ValidationDatabase. Lopuksi kutsutaan tehtävää ResetIIS, joka suorittaa komennon *iisreset* eRAServerillä, eli käynnistää IIS:n uudelleen.

4.4.3 Demoympäristön alustus

Demoympäristön alustuksessa ja käyttöliittymätestauksessa käytetään testiautomaatiokehystä nimeltä Robot Framework, joka soveltuu sekä hyväksymistestaukseen että hyväksymistestivetoiseen kehitykseen (Acceptance Test-Driven Development, ATDD). Se hyödyntää helppokäyttöistä tabulaattoripohjaista syntaksia, ja siinä voidaan suorittaa testitapauksia, jotka puolestaan voivat hyödyntää geneerisiä avainsanoja (keyword). Kehys on laajennettavissa Python- ja Java-pohjaisilla testikirjastoilla. Esimerkiksi kohdejärjestelmän testeissä käytetään kirjastoa nimeltään Selenium2Library, joka soveltuu web-sovellusten testaamiseen. Testitapauksia luodessa on mahdollista luoda korkean tason avainsanoja, jotka rakentuvat jo olemassa olevien avainsanojen päälle. [37]

InitializeDatabase on eRATest-virtuaalikoneella suoritettava tehtävä, jolla tietokannat alustetaan testausta varten. Se perustuu Robot Frameworkin robot-testitiedoston suorittamiseen. Alustuksessa suoritetaan kaikki testauksen kannalta tarpeelliset tehtävät, kuten käyttäjien ja koodistojen lisäämiset sekä ylläpitotehtävät. Vaikka Robot Framework on tarkoitettu hyväksymistestaukseen, se soveltuu myös web-käyttöliittymän manipulointiin muillakin tarkoituksilla, kuten tässä tapauksessa asetusten hallintaan käyttöliittymän kautta.

4.4.4 Käyttöliittymätestaus

Käyttöliittymätestauksessa suoritetaan eRAa varten luotuja käyttöliittymätestejä. Testit on suunniteltu siten, että niissä testataan yksityiskohtaisesti testin mukaisessa näkymässä esiintyviä elementtejä. Testit sisältävät usein myös HTTPS-rajapinnan yli kommunikointia muun muassa simulaattoreiden kanssa, joten käyttöliittymätestit soveltuvat osittain myös rajapintatestaukseen.

Ohjelmassa 4.1 on esimerkki robot-testitiedostosta, jossa on kaksi testitapausta sisäänkirjautumiselle. Testitiedoston alussa on asetukset, jossa muun muassa määritellään käytettävät kirjastot sekä resursseja ja muuttujia sisältävät tiedostot. Lisäksi asetuksissa määritellään testitapausta edeltävät ja sen jälkeiset toimenpiteet. Tiedostossa voidaan myös

```
*** Settings ***
Library                   Selenium2Library    run_on_failure=Nothing
Library                   ../Resources/keywordsLogin.py
Resource                  ../Resources/keywordsLogin.robot
Variables                 ../Resources/doctorRelatedVariables.py
Variables                 ../resources/commonVariables.py
Test Setup                Open Browser  ${site_url} ${browser}
Test Teardown             Run Keyword If Test Failed  Close Session
Suite Teardown            Close Browser
Force Tags                Login

*** Variables ***
${occupation_description} 001 laillistettu lääkäri
${degree_code}            76111-222 lääketieteen lisensiaatti

*** Test Cases ***
Cancelled Login
  Default Login Task
  Press When Displayed    buttonCancelLogin
  Wait Element            buttonLogin
  Close Browser

Login Independent User
  [Tags]                  IAH
  Default Login Task
  Press When Displayed    buttonLoginAsIndependentUser
  Validate User Informations  ${occupation_description} ${degree_code}
  Log User Out
```

Ohjelma 4.1. *Esimerkki Robot Frameworkin testitiedostosta, jolla tutkitaan sisäänkirjautumista.*

määritellä paikallisia muuttujia. Testitapauksille voidaan merkitä tunnisteita (tag), jolloin voidaan ajaa joukko testitapauksia tietyllä tunnisteella. Kuvan 4.1 esimerkkiohjelmassa ei ole erikseen luotu avainsanoja, mutta siinä käytetään *keywordsLogin.robot*-tiedostossa määriteltyjä avainsanoja, kuten esimerkiksi avainsanaa *Default Login Task*.

GUITest on eRATest-virtuaalikoneella suoritettava tehtävä, joka suorittaa käyttöliittymätestit järjestelmälle testauskategorioittain. Windowsilla testit ajetaan batch-komennolla *call robot*, jota tehtävässä kutsutaan jokaiselle testauskategorialle. Jokaisesta kategoriasta syntyy XML-pohjainen testausraportti, jotka testien ajamisen jälkeen yhdistetään yhdeksi XML-tiedostoksi käyttämällä Robot Frameworkin tarjoamaa työkalua nimeltä *rebot*. Lopuksi kutsutaan Pythonilla toteutettua raportointityökalua *reporter.py*, joka luo yhdistetyn XML-tiedoston perusteella kompaktin testausraportin PDF-muotoisena.

4.4.5 Rajapintatestaus

Rajapintatestauksessa käytetään yrityksellä kehitettyä rajapintatestaustyökalua nimeltään InterfaceTesterTool. Se on komentorivisovellus, joka suorittaa sille syötettyjä XML-pohjaisia testitiedostoja. Työkalu on haarautettu jo pidempään kehitetystä käyttöliittymäsovelluksesta nimeltään InterfaceTester, mutta kummatkin sovellukset käyttävät samoja tie-

```
<test>
    <description>Ping</description>
    <arrangements>
        <users>
            <user type="doctor" />
        </users>
        <organizations>
            <organization type="public_healthcare" />
        </organizations>
        <system_clients>
            <system_client type="organization" />
        </system_clients>
    </arrangements>
    <statement_list>
        <function_call>
            <name>Ping</name>
            <expected_result_list>
                <expected_result>
                    <status>200</status>
                </expected_result>
            </expected_result_list>
        </function_call>
    </statement_list>
</test>
```

Ohjelma 4.2. *Esimerkki rajapintatestitiedostosta, jolla testataan työkalun ja eRAn välistä yhteyttä.*

tomalleja. Haarautus oli käytännössä välttämätöntä, koska Jenkinsin kautta on huomattavasti kätevämpää suorittaa muutama batch-komento kuin manipuloida erillistä käyttöliittymäsovellusta testien ajamiseksi.

Ohjelmassa 4.2 on esitetty esimerkki yksinkertaisesta testitiedostosta, jonka avulla testataan työkalun ja eRAn välistä HTTPS-yhteyttä. Lohkossa *arrangements* asetetaan lähtötiedot tiedoston testeille, kuten käyttäjän, organisaation sekä asiakasjärjestelmän tyypit. Lohkon *statement_list* sisällä määritellään järjestyksessä suoritettavat konfiguraatiomuutokset ja funktiokutsut. Funktiokutsujen lohkoihin määritellään kutsuttavan funktion nimi ja listataan odotetut paluuarvot. Työkalu lähettää eRAlle HTTP POST -pyynnön funktion nimen perusteella, jolloin eRA käsittelee pyynnön ja palauttaa vastauksen työkalulle. Vastauksen paluuarvoa verrataan testitiedossa määriteltyihin odotettuihin paluuarvoihin, josta määräytyy funktiokutsun onnistuminen. Monien funktiokutsujen yhteydessä eRA lähettää HTTP-pyyntöjä simulaattoreille ja odottaa tiettyjä paluuarvoja. Täten yhden testiketjun suorituksen aikana voidaan testata montaa eri rajapintaa.

InterfaceTest on eRATest-virtuaalikoneella suoritettava tehtävä, joka suorittaa rajapintatestit järjestelmälle. Alla on esitetty tehtävässä ajettava batch-komento.

```
"%TOOL_PATH%" --testChainPath "%TESTCHAIN_PATH%" --configurationPath "%
    CONFIG_PATH%" --errorCodeDatabasePath "%ERROR_DATABASE_PATH%" --
    serverAddress https://192.168.2.12:44300/ --databaseERA "
    eRADatabase" --connectionStringERA "Data Source=192.168.2.12,1433;
    Initial Catalog={0};Integrated Security=SSPI;" --outputPath "%
    OUTPUT_PATH%"
```

Testaustyökalulle voidaan asettaa monia eri asetuksia, mutta Jenkinsin kautta ajettava tehtävä ei ole määritelty ottamaan parametreja vastaan. Työkalu etsii suoritettavia testitiedostoja syötetystä hakemistosta ja sen alihakemistoista. Hakemistossa voi esiintyä myös muitakin tiedostoja, jotka työkalu erottaa itse testien joukosta. Hakemistossa voi myös esiintyä testejä, joita ei kuulu ajaa jatkuvan toimituksen yhteydessä. Sellaisia testejä ovat esimerkiksi käyttäjän vuorovaikutusta vaativat testit sekä tulevia rajapintaversioita varten laaditut testit. Ohitettavat testitiedostot sisältävät kentät, joissa mainitaan syy testin ohittamiselle.

Konfiguraatiotiedostossa voidaan asettaa yleisiä asetuksia, joita kaikki suoritettavat testit käyttävät. Sellaisia ovat esimerkiksi testeissä hyödynnettävien palvelimien IP-osoitteet, testikäyttäjät sekä organisaatiot. Virhetietokanta on tiedosto, joka sisältää tietoa virhekoodeista, joita eRA voi palauttaa HTTP-vastauksissa. Lisäksi tiedosto sisältää ohjeita sille, kuinka työkalun pitää toimia kyseisten virheiden tapauksissa, koska eRAlta vastaanotettu virhe ei saa aina johtaa testin epäonnistumiseen. Komennolle voidaan antaa parametreina myös oletusosoite, käytettävän tietokannan nimi sekä yhteysmerkkijono, jonka avulla työkalu pystyy yhdistämään eRAn tietokantaan. Työkalu luo testaustulokset komennossa määriteltyyn polkuun. Testaustuloksina syntyy kattava lokitekstitiedosto sekä kompakti testausraportti PDF-muodossa.

5. ARVIOINTI JA JATKOKEHITYS

Toteutettua jatkuvan toimituksen liukuhihnaa on suoritettu säännöllisesti sen valmistumisesta lähtien. Liukuhihna on tuottanut melko yhteneväisiä tuloksia suorituskertojen kesken niin ajankäytön kuin testaustulosten osalta.

Liukuhihnan suorittamien vaiheiden kestot on saatu kerättyä Jenkinsin käyttöliittymästä. Testaustulokset on koostettu käyttöliittymätesteissä käytetyn Robot Frameworkin ja rajapintatestaustyökalun tuottamista testausraporteista.

5.1 Tulokset

Taulukossa 5.1 on esitetty jatkuvan toimituksen suorittavan Master-liukuhihnan vaiheisiin kuluneet ajat. Koska osa vaiheista ajettiin rinnakkain, kokonaisaika on pienempi kuin vaiheiden kestojen summa. Kestoista havaitaan, että liukuhihnan alkupään vaiheet vievät vähän aikaa, mutta demoympäristön alustaminen ja testaus vievät suurimman osan ajasta. Käyttöliittymätestaus vie vajaan 90 prosenttia koko liukuhihnan suorituksen kestosta, ja rajapintatestaus vajaan 10 prosenttia. Käyttöliittymätestaus vie noin yhdeksänkertaisen ajan rajapintatestaukseen nähden, vaikka käyttöliittymätestejä on vain noin kaksi kertaa enemmän kuin rajapintatestejä. Se selittyy sillä, että jokaisen käyttöliittymätestin aikana suoritetaan enemmän toimintoja, ja käyttöliittymän manipuloinnissa esiintyy viivettä automatiikan hyödyntämisestä huolimatta. Lisäksi virhetilanteista toipuminen vie käyttöliittymätesteissä enemmän aikaa, koska virheen havaitsemisessa on viivettä ja testin lopetus vaatii erillisten toimenpiteiden suorittamista.

Taulukko 5.1. *Master-liukuhihnan vaiheisiin kuluneet ajat.*

Vaihe	**Kesto**
eRA	2min 52s
SimArkisto	1min 40s
SimReseptikeskus	1min 30s
SimValvira	1min 24s
UpdateDataFiles	9s
InitializeGUITester	10s
InitializeInterfaceTester	1min 32s
DropDatabases	1min 28s
InitializeDatabase	24min 30s
InterfaceTests	57min 42s
GUITest	8h 50min
Master-liukuhihnan kesto	10h

Taulukossa 5.2 on esitetty käyttöliittymätestien tulokset. Testillä voi olla monta tunnistetta, eli testien yhteiskesto on lyhyempi kuin tunnisteittain tarkasteltavien testien kestojen

summa. Suurin osa epäonnistumisista johtuu ongelmasta, jossa SimArkisto väittää suoritetun allekirjoituksen olevan virheellinen. Virhe toistuu satunnaisesti eri testeissä demoympäristössä ajettaessa, mutta ongelma ei ilmene suoritettaessa testejä paikallisesti omalla koneella. Ongelmaa on tutkittu, mutta sitä ei ole toistaiseksi saatu ratkaistua. Suuri osa hammaspuolen testeistä epäonnistui käyttöliittymätestauksen aikana. Hammaspuolen käyttöliittymät ovat olleet viimeaikaisen kehityksen kohteena, jonka seurauksena kyseiset testit eivät ole pysyneet ajan tasalla. Kaiken kaikkiaan epäonnistuneiden testien määrä on kohtalaisen pieni, ja monet sekä niistä että eRAn ongelmista pitäisi olla melko nopeasti korjattavissa.

Taulukko 5.2. *Käyttöliittymätestien tulokset tunnisteittain.*

Tunniste	Yhteensä	Onnistuneet	Epäonnistuneet	Kesto (hh:mm:ss)
Kaikki	193	163	30	08:45:35
Archive	76	71	5	03:03:33
Certificate	21	21	0	00:54:20
Combination	7	7	0	00:33:50
Dental	36	18	18	02:23:26
Doctor	63	55	8	03:19:28
IAH	16	15	1	00:25:25
Login	12	12	0	00:02:52
MediNurse	30	30	0	01:20:49
Nurse	18	17	1	00:45:15
Osva	4	0	4	00:06:29
Patient	11	9	2	00:31:55
PracticalNurse	2	2	0	00:01:36
Prescription	9	9	0	00:49:59
Private	82	69	13	03:55:45
Private Doctor	17	11	6	00:22:13
Prohibition	9	9	0	00:10:28
Public	64	63	1	02:55:01
Renewal	4	4	0	00:01:12
RenewalTest	7	6	1	00:07:51
Student	30	25	5	01:23:13
Tab Visibility	1	1	0	00:00:52
Vaccination	1	1	0	00:03:05

Taulukoissa 5.3 ja 5.4 on esitetty rajapintatestien tulokset. Rajapintatesteistä epäonnistui noin puolet, mikä on kohtuuttoman paljon. Projektin rajapintatestaustyökalua aloitettiin kehittämään noin 2,5 vuotta sitten, mutta kehitys ei ole ollut kokoaikaista. Lisäksi työkalua on hyödynnetty neljän eri eRA-rajapintaversion kanssa, ja testit ovat osittain riippuvaisia rajapintaversiosta. Rajapintatestejä ei ole ajettu automatisoidusti ennen jatkuvan toimituksen liukuhihnaa, joten osa testeistä on huomaamatta muuttunut toimimattomiksi projektien edetessä. Testien epäonnistuminen ei ole itsessään liukuhihnan vika, vaan on pikemminkin arvokasta, että liukuhihna tuottaa täsmälliset testaustulokset säännöllisesti. Lisäksi projektien rajapinnoissa on lukuisia puutteita, eli monet rajapintatestit tuottavat virheitä ja varoituksia aiheellisesti.

Liukuhihnalla suoritetut tehtävät kestävät käytännössä yhtä kauan kuin paikallisessa kehitysympäristössäkin. Liukuhihnalla ei kuitenkaan esiinny merkittävää viivettä tehtävien vaihtumisen ohella, joten liukuhihnan suorittaminen kokonaisuudessaan on nopeampaa

Taulukko 5.3. *Rajapintatestien tulokset.*

Testit yhteensä	Onnistuneet testit	Epäonnistuneet testit	Testit varoituksilla	Ohitetut testit	Virheet	Varoitukset
103	46	38	39	19	2259	13763

Taulukko 5.4. *Rajapintatestien tulokset kategorioittain.*

Kategoria	Onnistuneet testit	Epäonnistuneet testit	Ohitetut testit	Virheet	Varoitukset
Archive	12	21	10	1072	8443
Authorization	0	2	0	502	0
Common and prescription	13	3	1	37	789
Dental	18	7	0	628	4525
eRAImageServer	0	0	4	0	0
Prescription	0	3	0	11	4
Security	1	0	2	0	0
Signed request	0	1	0	4	0
Test chains	1	0	2	0	2
Test command	1	1	0	5	0

kuin tehtävien suorittaminen putkeen manuaalisesti. Lisäksi liukuhihna voidaan käynnistää ajastetusti, jolloin se voidaan suorittaa säännöllisesti ilman turhaa vaivannäköä. Liukuhihna voidaan suorittaa työajan ulkopuolella, jolloin se ei vie resursseja keskellä työpäivää. Käytännössä työajan ulkopuolella suorittamisesta ei ole toistaiseksi merkittävää hyötyä, koska Jenkinsiä ei muutenkaan käytetä muihin tarkoituksiin työpäivän aikana. Jatkossa kuorman tasaaminen voi tulla tarpeeseen, mikäli ympäristössä aiotaan suorittaa muitakin toimenpiteitä.

Liukuhihnan mahdollistaman automaation myötä kattavaa käyttöliittymä- ja rajapintatestausta suoritetaan nykyään päivittäin, jolloin palvelun laadukkuutta pystytään arvioimaan säännöllisesti. Testien tulokset voivat poiketa toisistaan demoympäristön ja paikallisten kehitysympäristöjen välillä, jolloin testeistä voidaan löytää puutteita. Automaatiota hyödyntämällä koko testisarjaa ei tarvitse ajaa yhtä usein paikallisesti eri kehitysympäristöissä, mikä johtaa ajan ja resussien säästämiseen. Automaation ansiosta ohjelmointivirheet ovat löydettävissä nopeammin, jolloin ne voidaan myös korjata nopeammin.

5.2 Jatkokehitys

Tuloksien perusteella suurimpina heikkouksina havaittiin testauksen yksipuolisuus sekä epäonnistuneiden testien määrä. Kohdejärjestelmälle sovelletaan vain hyväksymistestauksen piiriin kuuluvia käyttöliittymä- ja rajapintatestausta. Järjestelmälle ei ole laadittu lainkaan yksikkötestejä, mikä on vakava puute. Vasta-argumenttina yksikkötestien puuttumiselle kohdejärjestelmä on erittäin rajapintariippuvainen, koska se toimii terveydenhuollon tietojärjestelmien ja Kelan Kanta-palveluiden integraatiorajapintana. Siitä huolimatta suuri osa järjestelmän ohjelmakoodista ei ole suoraan tekemisissä eri rajapintojen kanssa, joten ainakin kyseisiä osia pystyisi yksikkötestaamaan tälläkin hetkellä.

Jäljittelyn (mocking) avulla myös ulkoisia riippuvuuksia sisältävää koodia voitaisiin yk-

sikkötestata. Jäljittelyssä keskitytään testattavaan koodiin siten, että ulkoiset riippuvuudet korvataan olioilla, jotka matkivat ulkoisten riippuvuuksien käyttäytymistä [46]. Täten esimerkiksi yksikkötestauksessa kommunikaatio HTTPS-rajapinnan yli voitaisiin korvata logiikalla, jossa pyynnöt käsitellään omalla toteutustavalla. Lisäksi myös tietokantojen käyttöä voitaisiin jäljitellä aitojen käyttämisen sijaan. Yksikkötestaus voitaisiin suorittaa liukuhihnalla heti projektien koodien kääntämisen jälkeen. Siten koodin toimivuus saataisiin varmistettua ennen raskaita toimenpiteitä, ja koodin kattavuudesta saataisiin arvokasta tietoa.

Rajapintatestit ja eRAn integraatiorajapinnat ovat huomattavan kehittämisen tarpeessa, jotta epäonnistuneet testit saataisiin ratkaistua. Onnistuneen hyväksymistestauksen edellytys on, että epäonnistuneiden hyväksymistestien määrä saadaan pidettyä niin pienissä lukemissa, että epäonnistumiset kyetään tarkastelemaan lyhyessä ajassa ja perustamaan julkaisupäätös niiden varaan.

Jatkuvaa integraatiota ja toimitusta voitaisiin hyödyntää muissakin projekteissa. Esimerkiksi toimikortinlukijaohjelmisto eRASmartCardille on ehdotettu liukuhihnan toteutusta, jossa se voitaisiin kääntää eri alustoille koodin päivittymisen yhteydessä. Se olisi kätevästi toteutettavissa samalle Jenkinsin pääpalvelimelle, jolle jatkuvan toimituksen liukuhihna toteutettiin, ja se voisi muun muassa hyödyntää entuudestaan määriteltyjä tehtäviä. Skaalautuvuuden varmistamiseksi jatkossa joitakin tehtäviä täytyy muuttaa entistä geneerisimmiksi. Pääpalvelimen arkkitehtuuria ei kannata tosin hajauttaa liikaa ylläpidettävyyden vuoksi. Pääpalvelin jääkin todennäköisesti vain eRAan liittyvien projektien käyttöön, ja muille projekteille pystytetään omat ympäristöt.

Projektissa on periaatteessa mahdollista siirtyä soveltamaan jatkuvaa käyttöönottoa, jossa tuote julkaistaisiin automaattisesti tuotantoon. Käytännössä siirtyminen on kuitenkin epätodennäköistä, koska projekti vaatii tietoturvakriittisen luonteensa takia erittäin tarkkaa hyväksymistestausta, ja julkaisuihin liittyy paljon lainsäädännöllistä byrokratiaa.

6. YHTEENVETO

Työssä tutkittiin eri työkaluja ja menetelmiä jatkuvan toimituksen käyttöönotolle ohjelmistoprojektissa, jossa oltiin aiemmin noudatettu perinteistä toimitusprosessia. Perehtymisen jälkeen valittiin projektille parhaiten soveltuva työkalu, ja ympäristöä alettiin suunnittelemaan ja pystyttämään jatkuvalle toimitukselle. Lopputuloksena saatiin toteutettua liukuhihna, mikä suorittaa jatkuvan toimituksen vaiheet odotetulla tavalla. Taulukkoon 6.1 on koottu työn kannalta tärkeimmät tiedot.

Taulukko 6.1. *Yhteenvetotaulukko.*

Aihe	Jatkuva toimitus ohjelmistoprojektissa
Kohdejärjestelmä	eRA (Atostekin web-palvelu)
Työkalu	Jenkins
Eriyttämismenetelmä	Virtualisointi
Liukuhihnan keskimääräinen suoritusaika	10h
Ympäristön pystytyksen aika-arvio	500h
Käyttöliittymätestit	163 onnistunutta, 30 epäonnistunutta
Rajapintatestit	46 onnistunutta, 38 epäonnistunutta
Vahvuudet	Testausautomaatio, käännösten toistettavuus, säännölliset julkaisut
Heikkoudet	Ympäristöriippuvaisuus, testien yksipuolisuus ja epäonnistumismäärä, toistaiseksi vähäinen käyttö

Liukuhihnan ansiosta kääntämistä ja testausta saadaan suoritettua automatisoidusti toteutetussa ympäristössä. Testaajien ei enää tarvitse ajaa pitkiä testisarjoja paikallisesti omilla koneillaan, vaan he voivat hyödyntää liukuhihnan tarjoamia tuloksia. Kehittäjien työstämille muutoksille saadaan suoritettua laadunvarmistusta tiheään tahtiin, jolloin mahdolliset ongelmat ovat nopeasti havaittavissa. Automaation ansiosta projektissa säästetään henkilötyötunteja ja sen myötä kustannuksia.

Projektissa kehitettävää palvelua ei ole vielä julkaistu tuotantoon hyödyntämällä toteutettua liukuhihnaa. Palvelulla on jo lukuisia asiakkaita, ja palvelu on rajapintapainoitteisen luonteensa takia hyvin tietoturvakriittinen. Täten on ymmärrettävää, että tuote julkaistaan harvemmin, tällä hetkellä vain reilun kuukauden välein. Kuitenkin kyseiset julkaisut pitäisi perustaa liukuhihnan tuottamien testaustulosten ympärille, koska hyväksymistestien tarkoituksena on toimia kynnyksenä julkaisun hyväksymiselle. Etenkin liukuhihnalla epäonnistuneiden testien määrän perusteella julkaisujen hyväksymiset ovat kyseenalaisia tällä hetkellä.

Projektin sisällä pitäisi siirtyä noudattamaan DevOps-kulttuuria, jotta jatkuvaa toimitusta pystyttäisiin hyödyntämään sen täydellä potentiaalilla. Kulttuurimuutos on haastavaa, koska kohdejärjestelmää on vuosien ajan kehitetty siten, että kehitys ja projektikohtaiset

toimenpiteet on pidetty erillään toisistaan tiimin keskuudessa. Lisäksi ohjelmallista testausta ollaan alettu hyödyntämään vasta pari vuotta sitten, ja testien laatiminen on ollut vain muutamien työntekijöiden varassa.

Työssä näytettiin, että jatkuvan toimituksen toteuttamiselle on tarjolla lähes rajattomasti eri työkaluja ja menetelmiä, joiden valinta riippuu usein kohdejärjestelmän ja -ympäristön asettamista vaatimuksista. Työssä toteutettu liukuhihna soveltuu käytettäväksi vain kohdejärjestelmän yhteydessä, mutta siinä on kuitenkin noudatettu jatkuvan toimituksen periaatteita. Suunnitteluvaiheessa on tärkeintä tiedostaa eri vaihtoehdot toimituksen toteuttamiselle, joita työssä esiteltiin paneutumatta liikaa yksityiskohtiin.

Työssä havaittiin, että jatkuvalla toimituksella on suuri merkitys laadukkaiden ohjelmistojen säännöllisessä tuottamisessa. Jatkuva toimitus vaatii kuitenkin ympärilleen työkulttuurin, jossa projektitiimin jäsenet vastaavat tuotteesta tasapuolisesti. Jatkuvan toimituksen käyttöönotto voi olla haastavaa ohjelmistoprojektista riippuen, mutta pitkällä tähtäimellä se maksaa itsensä moninkertaisesti takaisin. Oikein sovellettuna jatkuva toimitus tukee projektitiimin työskentelyä ja johtaa sekä parempaan asiakastyytyväisyyteen että liiketoiminnan kasvuun.

LÄHTEET

[1] Anastasov, M. What's the Difference Between Continuous Integration, Continuous Deployment and Continuous Delivery? Saatavissa (viitattu 23.8.2018): https://semaphoreci.com/blog/2017/07/27/what-is-the-difference-between-continuous-integration-continuous-deployment-and-continuous-delivery.html

[2] ASP.NET. Get building. Saatavissa (viitattu 10.10.2018): https://www.asp.net/

[3] Atostek Oy. Atostek eRA. Saatavissa (viitattu 23.8.2018): http://era.atostek.com

[4] Atostek Oy. Kelan Kanta-palvelut nopeasti käyttöön. Saatavissa (viitattu 4.9.2018): https://www.atostek.com/era/era-ammattilaisille/

[5] Atostek Oy. Nopea integraatio Kelan Kanta-palveluihin. Saatavissa (viitattu 10.10.2018): https://www.atostek.com/era/era-jarjestelmatoimittajille/

[6] Best Continuous Delivery Software. Saatavissa (viitattu 30.8.2018): https://www.g2crowd.com/categories/continuous-delivery

[7] Brizeno, M. 5 Traits of a Good Delivery Pipeline. Saatavissa (viitattu 23.8.2018): https://www.thoughtworks.com/insights/blog/5-traits-good-delivery-pipeline

[8] CircleCI, The shortest distance from idea to execution. Saatavissa (viitattu 23.8.2018): https://circleci.com/product

[9] CircleCI REVIEW. Saatavissa (viitattu 23.8.2018): https://reviews.financesonline.com/p/circleci/

[10] Codeship, Ship your Apps with Confidence. Saatavissa (viitattu 23.8.2018): https://codeship.com

[11] Continuous Deployment. Saatavissa (viitattu 30.8.2018): https://www.agilealliance.org/glossary/continuous-deployment

[12] Continuous Integration. Saatavissa (viitattu 30.8.2018): https://www.thoughtworks.com/continuous-integration

[13] Continuous Integration. Saatavissa (viitattu 31.8.2018):
https://continuousdelivery.com/foundations/continuous-integration/

[14] Continuous Testing. Saatavissa (viitattu 23.8.2018):
https://continuousdelivery.com/foundations/test-automation/

[15] Crisp, J. Automated Testing and the Test Pyramid. Saatavissa (viitattu 15.10.2018):
https://jamescrisp.org/2011/05/30/automated-testing-and-the-test-pyramid/

[16] Docker. Docker Containerization Unlocks the Potential for Dev and Ops. Saatavissa (viitattu 23.8.2018):
https://www.docker.com/why-docker

[17] Docker. Drive secure automation and deployment at massive scale. Saatavissa (viitattu 23.8.2018):
https://www.docker.com/solutions/cicd

[18] Docker. Get Started, Part 1: Orientation and setup. Saatavilla (viitattu 10.9.2018):
https://docs.docker.com/get-started

[19] Earnshaw, A. Top Benefits of Continuous Delivery: An Overview. Saatavissa (viitattu 30.8.2018):
https://puppet.com/blog/top-benefits-of-continuous-delivery-an-overview

[20] Groovy. A multi-faceted language for the Java platform. Saatavissa (viitattu 4.9.2018):
http://groovy-lang.org/index.html

[21] Guckenheimer, S. What is Continuous Integration? Saatavissa (viitattu 23.8.2018):
https://docs.microsoft.com/en-us/azure/devops/what-is-continuous-integration

[22] Hallivuori, K. Jatkuva käyttöönotto auttaa oppimaan ja hyötymään ohjelmistomuutoksista tehokkaasti. Saatavissa (viitattu 4.9.2018):
https://solinor.fi/2014/02/19/jatkuva-kayttoonotto-auttaa-oppimaan-ja-hyotymaan-ohjelmistomuutoksista-tehokkaasti/

[23] Heller, M. What is Jenkins? The CI server explained. Saatavissa (viitattu 23.8.2018):
https://www.infoworld.com/article/3239666/devops/what-is-jenkins-the-ci-server-explained.html

[24] IIS. Saatavissa (viitattu 23.8.2018):
https://www.iis.net/

[25] Jenkins, Build great things at any scale. Saatavissa (viitattu 23.8.2018):
https://jenkins.io

[26] Jenkins Glossary. Saatavissa (viitattu 23.8.2018):
https://jenkins.io/doc/book/glossary/

[27] Jenkins Pipeline Syntax. Saatavissa (viitattu 23.8.2018):
https://jenkins.io/doc/book/pipeline/syntax/

[28] Jenkins Plugins. Saatavissa (viitattu 23.8.2018):
https://plugins.jenkins.io/

[29] Kelly. Web Development: What is Staging? Design and Development. Saatavissa (viitattu 23.8.2018):
https://www.commonplaces.com/blog/web-development-what-is-staging/

[30] Lee, J. Configuring Parameters for Web Package Deployment. Saatavissa (viitattu 17.10.2018):
https://docs.microsoft.com/en-us/aspnet/web-forms/overview/deployment/web-deployment-in-the-enterprise/configuring-parameters-for-web-package-deployment

[31] Leff, A. Rayfield, J.T. Web-Application Development Using the Model View Controller Design Pattern. p.118. Viitattu 26.10.2018

[32] Leszko, R. (2017). Continuous Delivery with Docker and Jenkins, 1st ed. Packt Publishing, GB. Viitattu 20.9.2018.

[33] MSBuild. Saatavissa (viitattu 23.8.2018):
https://docs.microsoft.com/fi-fi/visualstudio/msbuild/msbuild

[34] Novoseltseva, E. Top 10 Benefits of Docker. Saatavissa (viitattu 27.9.2018):
https://dzone.com/articles/top-10-benefits-of-using-docker

[35] Reifman, J. Codeship: Continuous Integration and Delivery Made Simple. Saatavissa (viitattu 24.8.2018):
https://code.tutsplus.com/tutorials/codeship-continuous-integration-and-delivery-made-simple-cms-23517

[36] Remote Desktop Protocol. Saatavissa (viitattu 23.8.2018):
https://docs.microsoft.com/en-us/windows/desktop/termserv/remote-desktop-protocol

[37] Robot Framework. Saatavissa (viitattu 27.8.2018):
http://robotframework.org/

[38] Rouse, M. Microsoft SQL Server Management Studio (SSMS). Saatavissa (viitattu 24.8.2018):
https://searchsqlserver.techtarget.com/definition/Microsoft-SQL-Server-Management-Studio-SSMS

[39] Rouse, M. Staging environment. Saatavissa (viitattu 24.8.2018):
https://searchsoftwarequality.techtarget.com/definition/staging-environment

[40] Shahin, M. Zahedi, M. Babar, M.A. Zhu, L. An empirical study of architecting for continuous delivery and deployment. Viitattu 30.10.2018.

[41] sqlcmd Utility. Saatavissa (viitattu 23.8.2018):
https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility?view=sql-server-2017

[42] Srinivasan, B.N. What Tools Do You Need for Continuous Delivery? Saatavissa (viitattu 23.8.2018):
https://dzone.com/articles/101-tools-for-continuous-delivery

[43] Sypolt, G. Choosing a CI-CD Tool. Saatavissa (viitattu 23.8.2018):
https://saucelabs.com/blog/choosing-a-ci-cd-tool

[44] Travis, Core Concepts for Beginners. Saatavissa (viitattu 23.8.2018):
https://docs.travis-ci.com/user/for-beginners

[45] Travis, Test and Deploy with Confidence. Saatavissa (viitattu 23.8.2018):
https://travis-ci.org

[46] Unit Testing for Software Craftsmanship. Saatavissa (viitattu 22.10.2018):
https://www.telerik.com/products/mocking/unit-testing.aspx

[47] Visual Studio IDE. Saatavissa (viitattu 10.10.2018):
https://visualstudio.microsoft.com/vs/

[48] Web Deploy 3.6. Saatavissa (viitattu 23.8.2018):
https://www.iis.net/downloads/microsoft/web-deploy

[49] What is Continuous Delivery? Saatavissa (viitattu 23.8.2018):
https://continuousdelivery.com

[50] What is DevOps? Saatavissa (viitattu 23.10.2018):
https://aws.amazon.com/devops/what-is-devops/

[51] What is Regression Testing? Saatavissa (viitattu 17.10.2018): https://smartbear.com/learn/automated-testing/what-is-regression-testing/

[52] VMWare, Workstation Pro. Saatavissa (viitattu 23.8.2018): https://www.vmware.com/products/workstation-pro.html

[53] Zaiku Group. Continuous Delivery In a Nutshell. Saatavissa (viitattu 7.9.2018): https://medium.com/@Zaiku/continuous-delivery-in-a-nutshell-29f4213dabda

LIITE A: MASTER-LIUKUHIHNA

```
pipeline {
  agent { label 'master' }
  stages {
    stage('Projects') {
      parallel {
        stage('eRA') {
          steps {
            build job: 'eRA'
          }
        }
        stage('SimArkisto') {
          steps {
            build job: 'SimArkisto'
          }
        }
        stage('SimReseptikeskus') {
          steps {
            build job: 'SimReseptikeskus'
          }
        }
        stage('SimValvira') {
          steps {
            build job: 'SimValvira'
          }
        }
        stage('Data files') {
          steps {
            build job: 'Update', parameters: [[$class: 'StringParameterValue', name: 'updatePath', value: 'C:\\
                eRAPublish\\DataFiles']]
          }
        }
        stage('eRAGUITest') {
          steps {
            build job: 'Update', parameters: [[$class: 'StringParameterValue', name: 'updatePath', value: 'C:\\
                eRAPublish\\eRAGUITest']]
          }
        }
        stage('InterfaceTester') {
          steps {
            build job: 'InterfaceTester'
          }
        }
      }
    }
    stage('DropDatabases') {
      steps {
        build job: 'DropDatabases'
      }
    }
    stage('InitializeDatabase') {
      steps {
        build job: 'InitializeDatabase'
      }
    }
    stage('Interface tests') {
      steps {
        build(job: 'InterfaceTest', propagate: false)
      }
    }
    stage('GUI tests') {
      steps {
        build job: 'GUITest'
      }
    }
  }
}
```

Ohjelma A.1. *Master-liukuhihnan toteutus.*

LIITE B: ERA-LIUKUHIHNA

```
pipeline {
  agent { label 'master' }
  options { skipDefaultCheckout() }
  environment {
    PROJECT_FILE_PATH = "\\\\vmware-host\\Shared Folders\\eRAPublish\\eRA\\branches\\2.4\\eRAInterface"
  }
  stages {
    stage('Update') {
      steps {
        build job: 'Update', parameters: [[$class: 'StringParameterValue', name: 'updatePath', value: 'C:\\eRAPublish
            \\eRA']]
      }
    }
    stage('Build') {
      agent { label 'eRABuild' }
      steps {
        build job: 'Build', parameters: [[$class: 'StringParameterValue', name: 'projectFilePath', value:
            PROJECT_FILE_PATH ], [$class: 'StringParameterValue', name: 'projectFileName', value: 'eRAInterface'], [
            $class: 'StringParameterValue', name: 'profile', value: 'Deployment Package (ArchiveDemo)']]
      }
    }
    stage('Deploy') {
      agent { label 'eRAServer' }
      steps {
        build job: 'Deploy', parameters: [[$class: 'StringParameterValue', name: 'project', value: 'eRAInterface'], [
            $class: 'StringParameterValue', name: 'profile', value: '\\\\vmware-host\\Shared Folders\\eRAPublish\\
            eRA\\branches\\2.4\\Installation\\eRAInterface\\ArchiveDemo']]
      }
    }
  }
}
```

Ohjelma B.1. *eRA-liukuhihnan toteutus.*