Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Kandidaatintyö

Ville Hartikainen

Ohjelmistoalan pk-yrityksen mobiilikehityksen jatkuva integraatio

Työn tarkastaja(t): Tutkijatohtori Ari Happonen

Työn ohjaaja(t): Tutkijatohtori Ari Happonen
Tuotantojohtaja Ilkka Toivanen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma

Ville Hartikainen

Ohjelmistoalan pk-yrityksen mobiilikehityksen jatkuva integraatio

Kandidaatintyö

2018

50 sivua, 2 kuvaa, 8 taulukkoa, 1 liite

Työn tarkastaja: TkT Ari Happonen

Hakusanat: jatkuva integraatio, mobiiliohjelmistokehitys, DevOps
Keywords: continuous integration, mobile application development, DevOps

Työn tavoitteena on tutkia, mitä on jatkuva integraatio ja kuinka optimoida sen toteutus työnkulun sujuvoittamiseksi erilaisten käytänteiden ja työkalujen avulla. Tässä työssä jatkuvaa integraatiota tarkastellaan mobiilikehitys- ja DevOps -kontekstissa ohjelmistoalan pk-yrityksen näkökulmasta. Työn empiirisessä osassa tutkitaan yhden jatkuvan integraation työkalun soveltuvuutta mobiilikehityskäyttöön, implementoimalla jatkuva integraatio React Native -kehysympäristöllä kehitettävään mobiilikehitysprojektiin.

Työn tuloksena tunnistettiin, että jatkuvan integraation optimaalisuuteen vaikuttavat teknisten tekijöiden lisäksi organisaatiokulttuuri ja toimintamallit. Lisäksi havaittiin, että jatkuva integraatio on yksi tärkeimmistä DevOps-menetelmistä. Työn empiirisessä tutkimuksessa tunnistettiin mobiilikehityksen jatkuvan integraation erityispiirteitä sekä todettiin tutkittavan työkalun soveltuvan mobiilikehityskäyttöön ja DevOps-menetelmien työkalupalettiin.

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Computer Science

Ville Hartikainen

Continuous integration of mobile applications in a software industry SME

Bachelor's Thesis

2018

50 pages, 2 figures, 8 tables, 1 appendix

Examiner: D. Sc. Ari Happonen

Keywords: continuous integration, mobile application development, DevOps

The aim of this thesis is to research continuous integration and how to optimize its implementation for streamlining of workflow with the help of different methods and tools. This thesis examines continuous integration in mobile development and DevOps context from a viewpoint of a small to medium sized software development company. Empirical research of the thesis focuses on evaluating the suitability of one continuous integration tool to mobile application development by implementing continuous integration in a React Native -mobile application project.

As a result of the thesis, it was observed that the optimization of continuous integration is affected by technical aspects as well as organizational culture and operating models. Additionally, it was observed that continuous integration is a fundamental part of DevOps. Characteristics of continuous integration in mobile application development were recognized as a result of empirical research. It was found out that the researched tool suits to mobile application development as well as to the DevOps-toolbox.

SISÄLLYSLUETTELO

SYMBOLI- JA LYHENNELUETTELO

APK	Android Application Package
CD	Continuous Delivery
CI	Continuous Integration
DevOps	Development & Operations
JS	JavaScript
XML	Extensible Markup Language
YAML	YAML Ain’t Markup Language

1 JOHDANTO

Tietotekniset laitteet ovat kehittyneet yhä mobiilimpaan muotoon. Näin ollen palveluntarjoajien on täytynyt laajentaa palveluitaan sekä tukeaan mobiilimarkkinoille. Mobiililaitteiden kompakti koko ja kosketusnäyttö mahdollistavat uusien innovaatioiden ja palvelujen kehittämisen. Mobiililaitteille kehitetään jatkuvasti uusia sovelluksia ja näiden sovellusten kehittäminen on merkittävää liiketoimintaa. Tässä työssä käsitellään mobiiliohjelmistokehitystä sujuvoittavaa sekä sen laatua parantavaa prosessia, jatkuvaa integraatiota (Duvall, Matyas & Glover 2007). Johdannossa taustoitetaan työn lähtökohdat, tavoitteet sekä kuvataan työn sisältö.

1.1 Työn tausta

Ketterän ohjelmistokehityksen perusperiaatteet, kuten laadukkaan ohjelmiston jatkuvan toimittamisen mahdollistaminen ja muuttuviin asiakasvaatimuksiin vastaaminen (Agile Alliance 2018), edellyttävät laadukkaita prosesseja koko ohjelmistotuotantoketjulta. Jatkuva integraatio (CI eli Continuous Integration) on suosittu prosessi ketterien ohjelmistokehityksen menetelmien keskuudessa (Bosch 2014, s. 107). Jatkuvan integraation prosessit ovat lähimmässä vuorovaikutuksessa ohjelmiston lähdekoodin kanssa ja näin ollen perusta koko ketjun toiminnalle.

Ketterien menetelmien mukaelma, DevOps (Development & Operations), pyrkii yhdistämään usein toisistaan erillisen kehitystyön ja tuotannon yhtenäiseksi ketjuksi (Ebert, Gallardo, Hernantes ja Serrano 2016, s. 94). Kehitystyö ja tuotannolliset prosessit pyritään automatisoimaan hyödyntämällä prosesseihin soveltuvia työkaluja (Atlassian 2018a). Hyödynnettävien DevOps-työkalujen kirjo on nykypäivänä laaja, joten on tärkeää kartoittaa sekä arvioida, mitkä työkalut soveltuvat organisaatioihin parhaiten.

Työn aiheen on määrittänyt ohjelmistoalan pk-yritys OCTO3 Oy. DevOps on oleellinen osa OCTO3:n ajatusmaailmaa ja toimintamallia, jota hyödyntämällä yritys vastaa ketterästi sopimusohjelmistokehityksen haasteisiin. Jatkuvaa integraatiota on toteutettu yrityksessä projektikohtaisesti. Työn sisältämä tutkimus kartoittaa mahdollisuuksia jatkuvan integraation toteuttamiseen mobiilikehitysympäristössä sekä arvioi sen vaatimien työkalujen soveltuvuutta nopeasti kasvavan pk-yrityksen mobiilikehityshankkeisiin.

1.2 Tavoitteet ja rajaukset

Tämän kandidaatintyön tavoitteena on selvittää, kuinka mobiilikehityksen jatkuva integraatio on mahdollista toteuttaa ja optimoida pk-yrityksessä erilaisten käytänteiden ja työkalujen avulla sekä esittää arvio valitun työkalun soveltuvuudesta yrityksen tarpeisiin. Työssä kartoitetaan mobiilikehitysympäristöön soveltuvia jatkuvan integraation työkaluja ja käytänteitä sekä kuvataan lyhyesti työkalun valintaprosessin vaiheet. Työn pääpaino on empiirisessä tutkimuksessa, johon on valittu yksi jatkuvan integraation työkalu. Työkalun soveltuvuus ohjelmistoalan pk-yritys OCTO3 Oy:n mobiilikehitystarpeisiin arvioidaan.

Työn tutkimuskysymys on seuraava:

> Kuinka optimoida jatkuva integraatio ohjelmistoalan pk-yrityksen mobiilikehityksessä työnkulun sujuvoittamiseksi erilaisten käytänteiden ja työkalujen avulla?

Tutkimuskysymyksen jäsentelyn avuksi valittuja lisätutkimuskysymyksiä ovat seuraavat:

> Soveltuuko valittu työkalu pk-yrityksen mobiilikehityksen tarpeisiin?
>
> Tukeeko työkalu muita DevOps-käytänteitä?

Jatkuvan integraation optimoinnilla tarkoitetaan tämän työn kontekstissa työkaluja ja toimintamalleja, jotka tehostavat sekä helpottavat jatkuvan integraation toteuttamista laajemman DevOps-kulttuurin osana. Jatkuvan integraation optimaalisuutta valitulla työkalulla sekä työkalun soveltuvuutta arvioidaan peilaamalla empiirisen tutkimuksen tuloksia kirjallisuuskatsauksesta kerättyyn aineistoon.

Jatkuva integraatio kuvataan työssä laajemman DevOps-kulttuurin osana, jossa versionhallinnassa sijaitsevan ohjelmiston koodikannan jokaisesta päivityksestä käynnistetään integraatioprosessi. Integraatioprosessi on täysin automatisoitu, ja se sisältää ohjelmiston koontiversioinnin (*build*), testaamisen, käyttöönoton sekä koodin analysoimisen. Viimeisenä vaiheena prosessin suorituksesta toimitetaan kehittäjille ja muille asianomaisille raportti.

1.3 Työn rakenne

Luvussa kaksi kuvataan, mitä on jatkuva integraatio ja mitä käytänteitä siihen kuuluu. Lähtötietoina kerrotaan jatkuvan integraation taustaa sekä sen vaikutuksia ohjelmistoprojekteihin. Luvun tarkoituksena on avata lukijalle jatkuvan integraation konteksti sekä riippuvuudet muihin ketterän ohjelmistokehityksen (*DevOps*) toimintamalleihin. Jatkuvaa integraatiota kuvataan ja analysoidaan mobiilikehityksen näkökulmasta. Kolmannessa luvussa kartoitetaan yleisesti kirjallisuuskatsauksen keinoin, mitä jatkuvan integraation mahdollistavia työkaluja on olemassa sekä mitä ne tekevät. Pääpaino on varsinaisten CI-työkalujen vaihtoehdoissa sekä toiminnallisuuksissa. Lisäksi sivutaan päätyökalun suoritukseen kytkeytyviä jatkuvaa integraatiota tukevia työkaluja, kuten ohjelmakoodin katselmointi- ja arviointityökaluja. Luvussa neljä kuvataan yleisesti ohjelmistotyökalun soveltuvuuden määritys kirjallisuuskatsauksen pohjalta. Lisäksi kuvataan OCTO3 Oy:n asettamat tekniset vaatimukset. Luvussa myös kuvataan empiirisen tutkimuksen suoritus valitulla jatkuvan integraation työkalulla. Empiirisessä tutkimuksessa tutkitaan valittua jatkuvan integraation työkalua mobiilikehitysprojektissa. Käyttökontekstissa on vahvana testausautomaatio ja soveltuvuus DevOps-kokonaisuuteen. Viidennessä luvussa esitellään valittu työkalu, kuvataan havainnot empiirisestä tutkimuksesta sekä esitetään arvio työkalun soveltuvuudesta ohjelmistoalan pk-yrityksen mobiilikehityksen tarpeisiin. Viimeisessä luvussa esitetään työn johtopäätökset vastaamalla esitettyihin tutkimuskysymyksiin työn kirjallisuuskatsauksen ja empiirisen tutkimuksen pohjalta. Lisäksi reflektoidaan kriittisesti, kuinka työn tutkimus sujui ja tunnistetaan työn aiheeseen liittyviä jatkotutkimuskohteita.

2 JATKUVA INTEGRAATIO OHJELMISTOTUOTANNOSSA

Martin Fowlerin (2006) määritelmä prosessista, jossa kehittäjät integroivat tuottamansa koodin päälinjaan (*master*) vähintään päivittäin ja jonka jälkeen automatisoidusti koontiversioidaan sekä testataan uusi ohjelmistoversio, lienee nykypäivänä soveltuvin kuvaus jatkuvan integraation perusajatuksesta. Tässä luvussa kuvataan jatkuvan integraation prosessi kokonaisuudessaan sekä sen vaikutukset ohjelmistotuotantoon. Esiteltävillä kypsyystasomalleilla kuvataan jatkuvan integraation ulottuvuuksia. Lisäksi käsitellään, miten jatkuva integraatio soveltuu osaksi mobiiliohjelmistokehitystä sekä DevOps-kulttuuria.

2.1 Prosessi

Jatkuvan integraation toteuttamiseen ohjelmistokehityksessä vaaditaan lähtökohtaisesti kolme asiaa: versionhallinta, automatisoitu koontiversiointi (*build*) sekä kehittäjätiimin sitoutuminen vaadittuun työskentelymalliin (Humble & Farley 2010, s. 56-57). Versionhallintajärjestelmät ovat vakioituneet ohjelmistokehitykseen ja ne mahdollistavat lähdekoodissa tapahtuvien muutosten hallinnoinnin. Versionhallinnan hyödyntäminen sujuvoittaa kehityksen työnkulkua ja ratkaisee monta ohjelmiston koodikannan hallinnointiin ja ylläpitoon liittyvää ongelmaa. (Atlassian 2018c) Koontiversiointi on prosessi, jossa ohjelmiston lähdekoodi käännetään (*compile*) suoritettavaan muotoon. Termiä voidaan käyttää myös laajemman prosessin kuvauksena, minkä vuoksi koontiversiointi voi sisältää myös testaamista ja analysointia. (Duvall et al. 2007, s. 67) Teknisesti jatkuva integraatio on näin ollen yleisten ohjelmistokehityksen työmenetelmien automatisointia.

Jatkuva integraatio vaatii teknisten muutosten lisäksi myös yksilötason toimintamallien muutoksia. Seuraava listaus sisältää työskentelymallin käytänteet, joihin jatkuvaa integraatiota hyödyntävien kehittäjien tulisi sitoutua (Duvall et al. 2007; Humble et al. 2010).

- Vie lähdekoodia versionhallintaan useasti
- Älä vie versionhallintaan rikkinäistä koodia
- Korjaa rikkinäiset koontiversiot välittömästi
- Kirjoita kattavia automaattitestejä

- Suorita paikallinen koontiversiointi
- Kaikkien testien ja tarkistusten tulee onnistua
- Pidä paikallinen kehitysympäristö kunnossa
- Pidä integrointiprosessi lyhyenä

Jatkuvan integraation edellyttämiin toimintamallien muutoksiin sopeutuminen voi edellyttää organisaatiokulttuurillisia muutoksia, minkä vuoksi edellä mainittujen käytänteiden omaksuminen sekä jatkuvan integraation tekninen implementoiminen osaksi omaa kehitysketjua, kannattaa suorittaa pienemmissä osakokonaisuuksissa. Tämä on etenkin huomioitava, jos kehitystiimissä ei ole aiempaa kokemusta jatkuvasta integraatiosta tai kehitysprojektin toteutus on jo pitkällä. (Duvall et al. 2007, s. 36)

2.2 Työvaiheet

Jatkuvan integraation työvaiheet ovat kääntäminen, testaus, analysointi, käyttöönotto ja raportointi. Kehitettävän ohjelmiston tyypistä riippuen jatkuva integraatio voi sisältää lisätyövaiheita kuten tietokantaintegroinnin. Työvaiheiden automaattinen suoritus on edellytys jatkuvalle integraatiolle. Integrointiprosessi voidaan aloittaa versionhallinnan yhteyteen luotavalla kytköksellä (*hook*), joka aloittaa prosessin jokaisesta koodinkannan päivityksestä tai ajastetusti. (Duvall et al. 2007) Ohjeellinen yläraja työvaiheiden suorituksen kokonaiskestolle on kymmenen minuuttia ja tästä ajasta yleensä suurimman osan vie testaaminen. Mitä nopeampi integraatiorutiini on, sitä nopeampi on suoritusvaste ja yleinen työnkulku. (Fowler 2006)

Kääntäjäohjelma muuntaa korkeamman tason ohjelmointikielellä kirjoitetun lähdekoodin laitteistolle suoritettavaan muotoon ja ilmoittaa mahdollisista virheistä lähdekoodissa. Riippuen ohjelmointikielestä, kääntäminen voi edellyttää muiden ohjelmien, kuten esikääntäjän ja linkittäjän, hyödyntämistä. (Aho, Lam, Sethi & Ullman 2006) Jatkuvan integraation käännösvaiheessa versionhallinnan lähdekoodista tuotetaan suoritettava ohjelma tai sen osa. Työvaihe koskee vain kehitysprojekteja, joissa hyödynnetään käännettäviä ohjelmointikieliä. Kompleksisen ohjelmiston kääntäminen voi olla hidas toimenpide ja sen suoritusaikaan ei välttämättä pääse vaikuttamaan, lukuun ottamatta

kääntämisessä hyödynnettävän laitteiston päivittämistä ja mahdollisia ohjelmointikielen tai sen kääntäjän tarjoamia optimointikeinoja.

Testaaminen mahdollistaa ohjelmiston virheiden ja vikojen havaitsemisen sekä sen laadun varmistamisen (Kumar & Mishra 2016). Merkittävä osa ohjelmistoprojektin kustannuksista muodostuu testauksesta. Automaattisten testien kirjoittaminen ja suorittaminen, testausautomaatio, parantaa testauksen laatua ja vähentää kustannuksia pitkällä aikavälillä, vaikkakin se lähtökohtaisesti vaatii enemmän rahallista investointia. (Polo, Reales, Piattini & Ebert 2013; Kumar et al. 2016) Testausautomaatio on kriittinen osa jatkuvaa integraatiota ja hyvin toteutettuna mahdollistaa parhaan hyödyn saamisen koko prosessista. Jatkuvan integraation testausvaiheeseen voidaan sisällyttää automaattisia yksikkö-, komponentti-, systeemi- ja funktionaalisia testejä. Jatkuvan integraation testausvaiheessa oleellista on tunnistaa eri testien suoritusajat ja suorittaa vain ne testikategorioiden testit, jotka yksittäisen integraatioprosessin aikarajoitteiden sisällä on mahdollista suorittaa. Esimerkiksi pitkäkestoiset systeemi- ja funktionaaliset testit voidaan irrottaa erilliseksi suoritukseksi, jotta niitä ei ajeta jokaisesta versionhallinnan muutoksesta. (Duvall et al. 2007) Testaaminen tulisi myös suorittaa tuotantoympäristöä vastaavassa ympäristössä vakioiduilla riippuvuuksilla ja konfiguraatioilla (Fowler 2006).

Ohjelmiston tarkastaminen ja analysointi kuuluvat oleellisesti ohjelmiston laadunhallintaan. Ihmisten suorittamat tarkastukset, kuten lähdekoodin vertaisarviot ja läpikäynnit, ovat manuaalisia toimenpiteitä, jotka eivät sovellu suoranaisesti jatkuvan integraation automaattisten suoritusten osaksi. Kehitysympäristöt sisältävät usein tarkistustoimintoja ja -työkaluja, jotka ilmoittavat projektin lähdekoodissa olevista virheistä ja ongelmista sekä muotoilevat lähdekoodia automaattisesti (All About Circuits 2018). Näiden tarkastamista ja analysointia tukevien toimintamallien ja työkalujen hyödyntämisen tukena voi olla jatkuva integraatio. Duvall et al. (2007) mukaan automaattiset laadunhallinnalliset tarkastukset, kuten lähdekoodianalyysi, ovat luontevasti jatkuvan integraation prosessin yhteyteen kytkettäviä toimenpiteitä. Analysointivaiheessa lähdekoodi voidaan tutkia staattisesti ja/tai dynaamisesti. Analysointityökaluille määritellyistä säännöstöistä riippuen, työkalut voivat paljastaa lähdekoodista syntaksi- tai arkkitehtuuririkkomuksia.

Käyttöönottovaiheessa ohjelmiston lähdekoodista on muodostettu jatkuvan integraation prosessin lopputuote. Lopputuote on automatisoidusti käännetty, testattu ja analysoitu uusi koontiversio viimeisimmästä lähdekoodin tilasta. Jos koontiversio on läpäissyt jatkuvan integraation, voidaan se välittää käyttöönottovaiheessa seuraaviin prosesseihin, kuten hyväksymistestauksen kautta ohjelmiston julkaisuputkeen. Prosessit voivat vaihdella organisaatio- ja ohjelmistokohtaisesti. (Duvall et al. 2007; Humble et al. 2010)

Raportointivaiheessa tieto jatkuvan integraation prosessin suorituksesta ja sen lopputulemasta toimitetaan valittua sähköistä kanavaa pitkin kaikille asianomaisille. Sähköinen kanava voi olla esimerkiksi sähköposti tai organisaation sisäinen pikaviestinväline. Raportoinnissa voidaan esittää yksityiskohtainen loki kaikista suorituksista ja/tai yksinkertaistettu grafiikka suorituksen pääkohdista, kuten lopputulema ja tärkeimpiä metriikoita. Automaattinen ja luotettava raportointivaihe on kriittinen suoraviivaisen työnkulun kannalta, sillä prosessin epäonnistuessa prioriteettina tulee olla koontiversion korjaaminen. (Duvall et al. 2007)

2.3 Kypsyystasomallit

Jatkuvaan integraatioon on edellä kuvatut ohjeelliset työvaiheet, mutta prosessi voi olla merkittävästi erilainen riippuen sen kypsyystasosta. Jatkuvalle integraatiolle ja toimittamiselle on esitetty eri organisaatioiden toimesta kypsyystasomalleja, joilla pyritään visualisoimaan organisaation jatkuvan integraation tila ja mahdolliset kehityskohteet. UrbanCode:n (2018) jatkuvan toimittamisen kypsyystasomallissa on neljä kategoriaa, koontiversiointi (*building*), käyttöönotto (*deploying*), testaaminen (*testing*) ja raportointi (*reporting*). Jokaisella kategorialla on viisi eri kypsyystasoa, pohja- (*base*), aloittelija- (*beginner*), keski- (*intermediate*), kehittynyt- (*advanced*) ja äärimmäinen (*extreme*) taso. UrbanCode:n malli kuvaa jatkuvan toimittamisen kypsyyttä pelkästään teknisten osa-alueiden avulla. InfoQ:n (2018) vastaavassa jatkuvan toimittamisen kypsyysmallissa on viides tekninen kategoria, suunnittelu- ja arkkitehtuuri. Lisäksi malli esittää kypsyystasot organisaatiokulttuurille. Organisaatiokulttuuri on oleellinen osa jatkuvia DevOps-käytänteitä, joten kypsyyttä on hyvä arvioida myös sen näkökulmasta. Mallien kypsyystasot soveltuvat pääosin myös jatkuvaan integraatioon kypsyyden arviointiin.

Taulukossa 1 koontiversioinnin pohjatasolla edellytetään, että koontiversiointi suoritetaan erillisellä koneella. Keskitasolla jokaisesta versionhallinnan muutoksesta tulisi käynnistää koontiversiointiprosessi ja riippuvuuksienhallinta tulisi olla kehittynyt. Kypsimmällä tasolla koontiversiointeja suoritetaan samanaikaisesti helposti toistettavissa virtuaaliympäristöissä. (UrbanCode 2018)

Taulukko 1. Koontiversioinnin kypsyystasot (UrbanCode 2018)

Base	Build Script Build Machine
Beginner	Self-Service Build Nightly Builds Build artifacts are stored
Intermediate	Build on Commit Dependency Repository Secured Config
Advanced	Triggered Builds Build Cluster
Extreme	Build from VM Snapshots Gated Commits

Käyttöönoton kypsyydessä (taulukko 2) pelkästään jatkuvan integraation osuutta on vaikea arvioida. Jatkuvassa integraatiossa käyttöönotto tapahtuu käytännössä testauksen ja jatkoprosessien, kuten jatkuvan toimittamisen, mahdollistamiseksi. Pohjatasolla manuaalisesta työstä on siirrytty hyödyntämään komentoskriptejä. Mallin keskivaiheilla edellytetään tarkempaa hallintaa käyttöönotossa eri ympäristöjen välillä. Ääripäässä versionhallinnan muutoksesta käynnistyy täysin automaattinen julkaisuprosessi tuotantoon saakka. (UrbanCode 2018)

Taulukko 2. Käyttöönoton kypsyystasot (UrbanCode 2018)

Base	Deployment Scripts
Beginner	Self-Service Deploy to Test Auto Deploy of build to 1st Env. Mostly standardized Deploys
Intermediate	Self-Service Deploy to Test and Prod Standard process across all environments
Advanced	Test Gated Automation Promotions Database Deployments Coordinated SOA / multi-tier deploys
Extreme	Continuous Deployment to Production

Testauksen kypsyysmallin (taulukko 3) pohjatasolla edellytetään testiautomaatiota. Pohjatasolta seuraaville tasoille etenemiseen edellytetään automaattitestauksen laajentamista sekä staattisten analyysien käyttöönottoa. Kypsimmillä tasoilla testikattavuus on oltava korkea ja testien tulee myös huomioida tietoturvallisuus. (UrbanCode 2018)

Taulukko 3. Testauksen kypsyystasot (UrbanCode 2018)

Base	Some Test Automation
Beginner	Significant test execution at build time
Intermediate	Some Static Analysis Automated Functional Tests run nightly
Advanced	High Code Coverage (expansive tests) Security scans Risk based manual testing
Extreme	100% Coverage

Raportoinnin kypsyysmallin (taulukko 4) pohjatasolla kehittäjä tutustuu itsenäisesti työkalujen tuottamiin raportteihin. Keskitasolle eteneminen edellyttää raporttien saatavuuden laajentumista kaikille ohjelmistotuotannollisen ketjun ryhmille. Kehittyneimmillä tasoilla raporteista tuotetaan trendejä ja niitä hyödynnetään analysoimalla. (UrbanCode 2018)

Taulukko 4. Raportoinnin kypsyystasot (UrbanCode 2018)

Base	Visibility: Report Runner Tool generated reports
Beginner	Visibility: Team Latest reports always accessible
Intermediate	Visibility: Cross Silo Historical Reports available for viewing
Advanced	Report trending
Extreme	Cross Silo Analysis

Organisaatiokulttuurin (taulukko 5) pohjatasolla edellytetään työn jaottelua, dokumentaatiota ja tiheitä versionhallinnan koodikannan päivityksiä. Keskitasolle eteneminen edellyttää ketterien kehitysmenetelmien hyödyntämistä, tiimien välisten rajojen rikkomista sekä prosesseja. Kypsimmillä tasoilla kehitystiimit työskentelevät ristiin, työkaluille on vastuutiimi ja vastuuta jaetaan kehitystiimien kesken. (InfoQ 2018)

Taulukko 5. Organisaatiokulttuurin kypsyystasot (InfoQ 2018)

Base	Prioritized work Defined and documented process Frequent commits
Beginner	One backlog per team Share the pain Stable teams Adopt basic Agile methods Remove boundary dev & test
Intermediate	Extended team collaboration Component ownership Act on metrics Remove boundary dev & ops Common process for all changes Decentralized decisions
Advanced	Dedicated tools team Team responsible all the way to prod Deploy disconnected from Release Continuous Improvement (Kaizen)
Extreme	Cross functional teams No rollbacks (always roll forward)

Suunnittelun- ja arkkitehtuurin (taulukko 6) pohjatasolla edellytetään vakiintumista joihinkin alustoihin sekä teknologioihin. Keskitasoille eteneminen edellyttää versionhallinnassa haarautumisen minimointia ja siirtymisestä modularisaation hyödyntämisestä itsenäisten ohjelmistokomponenttien kehittämiseen. Kypsimmillä tasoilla arkkitehtuuri on viety äärimmäisyyksiin, mahdollistaen ympäristöjen pystyttämisen ja konfiguroinnin vähällä manuaalisella työllä. (InfoQ 2018)

Taulukko 6. Suunnittelun- ja arkkitehtuurin kypsyystasot (InfoQ 2018)

Base	Consolidated platform & technology
Beginner	Organize system into modules API management Library management Version control DB changes
Intermediate	No (or minimal branching) Branch by abstraction Configuration as code Feature hiding Making components out of modules
Advanced	Full component based architecture Push business metrics
Extreme	Infrastructure as code

2.4 Edut ja haasteet

Duvall et al. (2006) esittävät jatkuvan integraation toteuttamisen eduiksi riskien ja toistuvan manuaalisen työn minimoimisen, ohjelmiston käyttöönoton helpottumisen sekä projektin näkyvyyden ja tuotteeseen luottamuksen parantumisen. Riskejä minimoivat jatkuva testaaminen ja analysointi, jonka avulla ohjelmiston virheet ja ongelmat ovat havaittavissa aikaisessa vaiheessa sekä laatu on jatkuvassa tarkkailussa. Toistuvan manuaalisen työn minimoiminen vapauttaa kehittäjät keskittymään oleelliseen ja työn automatisointi takaa sen, että prosessit suoritetaan aina samalla tavalla. Jatkuva integraatio helpottaa huomattavasti ohjelmiston käyttöönottoa, mahdollistaen ideaalitapauksessa sen siirtämisen toimitusputkeen milloin tahansa jatkuvan toimittamisen prosessien avulla. Projektin näkyvyyttä ja päätöksentekoa tukevat jatkuvan integraation suorituksista kerättävä tieto ja trendit. Lisäksi luottamus tuotteeseen paranee, sillä testaus ja analysointi suoritetaan jokaisesta lähdekoodikannan muutoksesta.

Kyselytutkimuksessa, jota varten haastateltiin neljän eri ohjelmistotuotteen parissa työskenteleviä asiantuntijoita, havaittiin jatkuvan integraation parantavan projektin ennustettavuutta sekä projektihenkilöstön kommunikaatiota. Viitteitä havaittiin myös

testauksen tukemisesta ja kehittäjän tuotteliaisuuden paranemisesta. Tutkimustulokset kuitenkin osoittavat hajonnallaan sen, että jatkuvan integraation hyödyt voivat vaihdella hyvin paljon eri ohjelmistotuotteiden ja niiden kehitysprojektien välillä. (Ståhl et al. 2013) Hilton, Tunnell, Huang, Marinov ja Dig (2016) esittävät laajaan avoimen lähdekoodin ohjelmistoprojekteista kerättyyn aineistoon perustuen, että jatkuvaa integraatiota hyödyntävät projektit julkaisevat ohjelmistoaan useammin, hyväksyntäprosessi ehdotetuille lähdekoodimuutoksille on nopeampi ja päälinjaan ei liitetä ohjelmiston rikkovaa lähdekoodia.

Jatkuvan integraation implementoiminen edellyttää osaamista sekä resursseja. Avoimen lähdekoodin ohjelmistoprojekteissa merkittävimmiksi syiksi, miksi jatkuvaa integraatiota ei käytetä, havaittiin kehittäjien kokemattomuus, automaattitestien puuttuminen sekä se, ettei koodimuutoksia viedä versionhallintaan riittävän usein (Hilton et al. 2016, s. 432). Shahin et al. (2017, s. 16) toteuttamassa kirjallisuustutkimuksessa havaittiin vastaavasti hidasteena testaukseen liittyvät puutteet ja koodin liitosristiriidat (*merge conflicts*). Nämä haasteet indikoivat vahvasti, että jatkuvalla integraatiolla on merkittävä vaikutus työnkulkuun ja sen implementointiin vaikuttavat hyvin paljon ohjelmistotuotteen tai sen tuotekehitysprojektin tyyppi.

Havainnot tutkimusaineistojen pohjalta tukevat väitettä siitä, että jatkuva integraatio tarjoaa lukuisia etuja ohjelmistoprojektien työnkulkuun. Etuja ei voi kuitenkaan yksiselitteisesti yleistää kaikkiin ohjelmistoprojekteihin. Työn tutkimuskysymyksen kannalta tuleekin huomioida, etteivät havainnot jatkuvan integraation soveltuvuudesta työn pk-yrityksen mobiilikehityskontekstiin ole välttämättä suoraan sovellettavissa toisiin organisaatioihin tai kehitysprojekteihin.

2.5 Mobiilikehityksen osana

Mobiilisovellus on ohjelmisto, joka on suunniteltu suoritettavaksi mobiililaitteella, kuten älypuhelimella tai tabletilla. Mobiilisovellusten pääasialliset kehitysalustat ovat Android- ja iOS-käyttöjärjestelmät, sillä lähes kaikki mobiililaitteet hyödyntävät edellä mainittuja käyttöjärjestelmiä (NetMarketShare 2018). Mobiilisovellusten alustat ovat jatkuvassa muutoksessa. Uusia laitteita ilmestyy jatkuvasti ja käyttöjärjestelmäversiot päivittyvät vuosittain. Alustamuutoksiin reagoiminen sekä sovelluskaupoissa menestyminen

edellyttävät tiheää julkaisusykliä. Uusien ominaisuuksien ja virhekorjausten ketterä julkaisu vaativat kehitysketjulta nopeaa reagointia. Kehitykseen on näin ollen luontevaa hyödyntää jatkuvaa integraatiota. Mobiilisovelluksia voidaan kehittää natiivisti (eli alustakohtaisesti) sekä alustariippumattomasti (*cross-platform*). Natiivisovelluksia kehitetään alustan mukaisilla kehitystyökaluilla, rajapinnoilla sekä ohjelmointikielillä. Alustariippumattomasti mobiilisovelluksia voi kehittää selainpohjaisesti web-teknologioilla tai esimerkiksi React Native ja Xamarin -työkaluja hyödyntäen. (Jobe 2013, s. 27)

Mobiilikehityksessä suurimmiksi haasteiksi on havaittu muun muassa saman ohjelmiston ja käyttöliittymän kehittäminen usealle eri käyttöjärjestelmälle ja laitteistolle. Testaaminen on myös yksi mobiilikehityksen haasteista (Joorabchi, Mesbah & Kruchten 2013, s. 23; Wassermann 2013). Nämä haasteet kohdataan myös jatkuvaa integraatiota implementoidessa mobiilikehitysprojektissa. Natiivikehityksessä koodikanta on jokaiselle tuettavalle alustalle erilainen ja näin ollen jokaiselle alustalle tulee toteuttaa oma jatkuva integraatio. Alustariippumattomassa kehityksessä sen sijaan koodikannasta suurin osa on yhteistä kaikille alustoille. Mobiilisovellusten testaamisen haasteita ovat sen monimutkaisuus ja laitteiden sekä niiden ohjelmistoversioiden moninaisuus (Akour, Falah, Ahmad, Bouriat & Alemerien 2016). Mobiilikehityksen jatkuvan integroimisen optimoiminen edellyttää näin ollen alustojen asettamien lainalaisuuksien tuntemista sekä valittua kehitysteknologiaa tukevia ja ajantasaisia työkaluja.

2.6 DevOpsin osana

DevOps:in pääidea on ohjelmiston toimittamisketjun tehostaminen. Kehityspään (*development*) prosessi, jatkuva integraatio, ei yksinään kykene tähän, vaan tarvitaan sitä tukevia tuotannollisia (*operations*) prosesseja. (Virmani 2015, s. 78) Vaikka DevOps-termille ei ole varsinaista määritelmää (Roche 2013, s. 40), käsitetään termi yleisesti menetelmien ja työkalujen hyödyntämisenä, jotka edesauttavat ja automatisoivat ohjelmiston kehityksen ja tuotannon prosesseja (Atlassian 2018a; Ebert et al. 2016; OCTO3 2017). Jatkuva integraatio on yksi DevOps-toiminnan mahdollistava menetelmä.

Jatkuva toimittaminen (CD eli Continuous Delivery) on DevOps-menetelmä, joka laajentaa jatkuvaa integraatiota. Jatkuvan toimittamisen tavoitteena on mahdollistaa uuden ohjelmistoversion julkaisemisen tuotantoon suoraviivaisesti hetkenä minä hyvänsä

(Lianping 2015; OCTO3 2017). Oleellisia CD-prosesseja ovat mm. konfiguraationhallinta, hyväksymistestaus ja julkaisunhallinta (*release management*) (Humble et al. 2010). Jatkuvan integraation läpäisevä koontiversio voidaan tarvittaessa lähettää automatisoituihin jatkuvan toimittamisen prosesseihin, jonka lopputuotteena on ohjelmisto, joka on valmis julkaistavaksi.

DevOps-menetelmien ja työkalujen hyödyntäminen ohjelmistokehityksessä edellyttävät teknisiä ja yrityskulttuurillisia muutoksia. DevOps-menetelmien, kuten jatkuvan integraation, implementoiminen edellyttää työkalujen hyödyntämistä ja näiden työkalujen käyttöönotto voi vaatia muutoksia ohjelmistokehityshankkeen läpiviemiseen projektinhallinnan tasolta kehittäjän tasolle saakka (Zhu, Bass & Champlin-Scharff 2016). DevOps-menetelmien hyödyntämisen eduiksi on havaittu julkaisutiheyden ja testiautomaatio-osaamisen parantuminen sekä sen hyödyntäminen laajemmassa mittakaavassa. Organisaatiotasolla DevOps-menetelmät voivat tarjota parannusta kommunikaatioon sekä sallia kokeilevaa ohjelmistokehityskulttuuria. Merkittävimmät haasteet DevOps-kulttuurin hyödyntämiseen ovat kommunikaatio-ongelmat kehitys- ja tuotantotiimin välillä sekä organisaatiokulttuurimuutokset. (Riungu-Kalliosaari 2016)

3 JATKUVAN INTEGRAATION TYÖKALUT

Jatkuvan integraation toteuttamiseen hyödynnetään yleensä päätyökalua, ns. palvelinta, jonka tehtävä on kuunnella versionhallinnan koodikannassa tapahtuvia muutoksia ja aloittaa havaitusta muutoksesta koontiversiointiprosessi komentoskriptin mukaisesti sekä lopuksi toimittaa loki suorituksen kulusta kehittäjille (Meyer 2014, s. 15). Tähän päätyökaluun voidaan kytkeä aputyökaluja, jotka tukevat tai laajentavat jatkuvan integraation prosesseja sekä niiden suorittamista.

3.1 Jatkuvan integraation päätyökalut

Jatkuvaa integraatiota voi toteuttaa myös omilla koontiversiointiskripteillä, ilman varsinaista kyseiseen tarkoitukseen suunniteltua ohjelmistoa. Useat jatkuvan integraation työkalut ovat kuitenkin ilmaisia avoimen lähdekoodin ohjelmistoja ja työkalut suoraviivaistavat jatkuvan integraation implementointia. (Duvall et al. 2007 s. 8) Pienikustanteinen tai jopa täysin ilmainen ohjelmistoratkaisu, joka todennäköisesti tuo lisäarvoa ohjelmistoprojektiin, on hyvin houkutteleva konsepti. Täytyy kuitenkin huomata, että työkalusta riippuen varsinainen hankintakustannus voi olla hyvin minimaalinen verrattuna työkalun ylläpito- ja konfigurointikustannuksiin.

Jatkuvan integraation päätyökalun perustoiminnallisuuksiin kuuluvat versionhallinnan kuuntelu, koontiversiointiskriptien suoritus sekä suorituksen kirjaaminen lokiin, koontiversion leimaaminen ja ilmoitusten lähetys. Työkalun tulee mahdollistaa ohjelmistokoodin kääntäminen, riippuvuuksien hallinta, testien suorittaminen ja koontiversion käyttöönotto. Työkalu voi tarjota käyttöliittymän toiminnallisuuksien konfigurointiin sekä raporttien esittämiseen. Jatkuvan integraation työkalujen hyödyntämisestä on luontaista siirtyä automatisoimaan myös ohjelmiston julkaisua, mihin hyödynnettävä työkalu voi tarjota mahdollisuuden suoraan tai tarjota hyvän rajapinnan koontiversion käyttöönottoon julkaisuketjussa.

Kirjallisuudesta, konferenssijulkaisuista ja artikkeleista löytyy kuvauksia sekä tutkimuksia jatkuvasta integraatiosta ja sen prosesseista yleisellä tasolla. Sen sijaan vain muutamia työkaluja on käsitelty kirjallisuudessa. Työkalujen tutkimuksen suhteen pääasiallisia lähteitä ovat verkkoartikkelit ja vastaavanlaiset julkaisut. Työkalujen tarjoajien internetsivustot,

tuotekuvaukset ja dokumentaatiot riittävät pääosin jatkuvan integraation implementoimiseen. Implementaation soveltuvuus sekä optimaalisuus käyttöympäristöön varmistetaan kuitenkin vain tutkimalla ja testaamalla.

Jatkuvan integraation päätyökalujen suosiosta löytyy kattava arviointi G2 Crowd -verkkosivustolta. Ohjelmistoalalla työskentelevät tai alalla vaikuttavat henkilöt voivat käydä arvioimassa sivustolle käyttämiään jatkuvan integraation työkaluja (G2Crowd 2018). Näin ollen sivuston tilastoista saa melko relevanttia sekä ajantasaista tietoa siitä, mitä työkaluja käytetään teollisuudessa, ja mitkä niiden vahvuudet sekä heikkoudet ovat. Viisi parhaaksi arvioitua ja johtavan markkina-aseman omaavia jatkuvan integraation työkalua ovat CircleCI, Travis, CodeShip, Jenkins ja TeamCity (G2Crowd, 2018). Näistä työkaluista Jenkins on ilmainen avoimen lähdekoodin ohjelmisto ja muut tarjoavat ilmaisen kokeilujakson rajatuilla ominaisuuksilla. Ilmainen kokeilujakso on hyödyllinen etenkin, jos kokee epävarmuutta työkalun lähtökohtaiselle soveltuvuudelle hyödynnettävien kehitysteknologioiden kanssa. Jos kehitysprojekteja on useita tai kehitysprojekti on valtavan laaja, saattavat samanaikaisuuteen ja koontiversiointien lukumääriin asetetut rajat rajoittavat työskentelyä. Työkalun toimittajasta riippuen löytynee kuitenkin kaikkiin käyttötapauksiin sopiva ja skaalautuva ratkaisu maksua vastaan. Huomionarvoista on se, että maksulliset työkalut tarjoavat avoimen lähdekoodin kriteerit täyttäville ohjelmistokehitysprojekteille yleisesti pienemmät kustannukset tai jopa ilmaisen palvelun.

Jatkuvan integraation työkalun valinnassa huomioitavia asioita ovat työkalun tarjoamat toiminnallisuudet, soveltuvuus käyttöympäristöön, luotettavuus, pitkäikäisyys ja käytettävyys (Duvall et al. 2008, s. 248-255). Käyttöympäristöön soveltuvuuden perusedellytys kaupallisissa jatkuvan integraation työkaluissa on yhteensopivuus käytettävän versionhallintajärjestelmän kanssa. Lisäksi on huomioitava, haluaako käyttää avoimen- vai suljetun lähdekoodin työkalua ja ajetaanko integrointirutiini ulkopuolisella vai omalla palvelimella. Työkalun käytön oppimiskäyrän jyrkkyys ja dokumentaation sekä tuen taso ovat myös tärkeitä kartoituksen kohteita. (Sqreen 2018) Ohjelmistotyökalun valinnassa oleellista on myös kartoittaa, löytyykö verkosta tai kirjallisuudesta materiaalia vastaavan käyttötapauksen toteuttamiseen.

3.2 Jatkuvan integraation aputyökalut

Jatkuvan integraation aputyökaluja ovat työkalut, joita voidaan kytkeä päätyökalun yhteyteen ja jotka tukevat jatkuvan integraation prosesseja. Aputyökaluina voidaan hyödyntää esimerkiksi staattisia lähdekoodianalysointityökaluja, jotka käyvät koodikantaa läpi ja ilmoittavat työkalulle määriteltyjen koodauskäytänteiden rikkomisesta. Staattinen lähdekoodianalyysi on kriittinen osa laadukasta testausta (Wang, Meng, Han & Bai 2014, s. 1280). Sen mahdollistavat työkalut tukevat manuaalista koodikatselmointia ja nopeuttavat jatkuvan integraation prosessia (Duvall et al. 2007, s. 163). SonarQube on yksi kaupallinen analyysi- ja laadunhallintatyökalu, joka soveltuu osaksi DevOps-käytänteitä ja se voidaan integroida CI-työkaluun (SonarQube 2018). Eri ohjelmointikielille saatavat lint-työkalut mahdollistavat staattisen lähdekoodianalyysin (Gimpel 2014). Riippuen ohjelmistoprojektin tyypistä, kattavan testiautomaation suorittaminen voi vaatia lisäksi erillisien testauskehysympäristöjen (*framework*) hyödyntämistä. Testiautomaatiota tukevia aputyökaluja voidaan ja on suotavaa hyödyntää jatkuvan integraation yhteydessä, jotta kaikki paikallisesti suoritetut testit voidaan suorittaa myös integraatioympäristössä. Aputyökalujen hyödyntämiselle tarjottava tuki riippuu päätyökalusta.

3.3 Mobiilikehitykseen soveltuvat työkalut

G2Growdin (2018) viidestä parhaaksi arvioidusta ja johtavan markkina-aseman omaavista jatkuvan integraation työkaluista natiivimobiilikehitystä (iOS ja Android) tukevat virallisesti CircleCI ja Travis (CircleCI 2018; Travis 2018). Muita mobiilikehityksen jatkuvaa integraatiota tukevia työkaluja ovat muun muassa Bitrise, GitLab CI ja Nevercode (Bitrise 2018; GitLab CI 2018; Nevercode 2018). Lisäksi Bamboo- ja Jenkins-työkaluille on lisäosia, joilla saavutetaan tuki mobiilikehitykselle (Atlassian 2018b; Jenkins 2018). Tuki alustariippumattomille kehitysteknologioille täytyy selvittää työkalukohtaisesti. Työkalujen markkinoinnista ja dokumentaatiosta ei välttämättä saa yksikäsitteistä tietoa, tuetaanko alustariippumattomia teknologioita. Mobiilikehitysprojektissa voidaan hyödyntää lukuisia erilaisia aputyökaluja ja kehysympäristöjä. Staattinen lähdekoodianalyysi mobiilikehityksen jatkuvan integraation osana voi edellyttää aputyökalujen hyödyntämistä, jotka tarjoavat tuen käytettävälle ohjelmointikielelle sekä integroituvat päätyökaluun. Esimerkiksi natiivikehitysympäristöt Android Studio ja iOS-käyttöjärjestelmän Xcode sisältävät työkalut staattiseen lähdekoodianalyysiin (Android Studio 2018; Apple 2018). Mobiilikehityksessä

suosittuja testiautomaatiotyökaluja ovat muun muassa Appium, Espresso ja Calabash (Astegic 2018).

4 OHJELMISTOTYÖKALUN SOVELTUVUUS

Työn tutkimuskysymykseen vastaaminen edellyttää asetettuihin lisätutkimuskysymyksiin vastaamista. Tässä kappaleessa kuvataan ohjelmistotyökalun soveltuvuuden teoriaa, asetetaan soveltuvuuden vaatimukset sekä kuvataan toteutettava empiirinen tutkimus.

4.1 Ohjelmistotyökalun soveltuvuuden arviointi

Ohjelmistotyökalun soveltuvuuden arviointiprosessi koostuu neljästä vaiheesta. Vaiheet ovat arviointikriteeristön kehittäminen, kokeiden määrittäminen, kokeiden suorittaminen ja tulosten analysointi (Firth, Mosley, Pethia, Roberts & Wood 1987, s. 33-34). Soveltuvuuden arviointiprosessi edellyttää näin ollen määriteltyjen vaatimusten listaamista sekä vaatimusten täyttävien raja-arvojen määrittämistä. Jokaiselle vaatimukselle tulee määrittää todennustapa. Todennustapa voi koostua yhdestä tai useammasta kokeesta, joiden perusteella voi todeta analysointivaiheessa vaatimuksen raja-arvon täyttyvän tai alittuvan.

Tässä työssä työkalun soveltuvuuden arviointiin hyödynnetään ohjelmiston laadun arvioimiseen tarkoitettua ISO/IEC 25010 -standardia sekä OCTO3 Oy:n asettamia vaatimuksia. Arvioinnissa tarkastellaan myös vastaavuutta jatkuvan integraation määritelmän mukaiseen prosessiin sekä soveltuvuutta DevOps-malliin. ISO/IEC 25010 -standardissa määritellään kriteerit ohjelmiston laadun arviointiin. Pääkriteerejä on kahdeksan, joita kuvaavat tarkemmin yhdestä kuuteen alakriteeriä. Pääkriteerit ovat toiminnallinen sopivuus, tehokkuus, yhteensopivuus, käytettävyys, luotettavuus, turvallisuus, ylläpidettävyys ja siirrettävyys. (ISO/IEC 2011) Työkalun laatua voidaan arvioida näiden kriteerien pohjalta. Työkalun laadun arvioiminen tukee soveltuvuuden arviointiprosessia. Työkalun tulee mahdollistaa jatkuvan integraation määritelmän sisällön toteuttaminen. Sisältöön kuuluvat lähdekoodin kääntäminen, testaus, analysointi, käyttöönotto ja raportointi (Duvall et al. 2007, s.12-20).

4.2 Vaatimukset ja raja-arvot

OCTO3 Oy:n asettama viitekehys jatkuvan integraation työkalun soveltuvuuden määrittämiselle on seuraavanlainen. Työkalun tulee mahdollistaa React Native -kehysympäristöllä toteutettavan mobiiliohjelmistokehityksen jatkuva integraatio. Jatkuvan integraation ympäristön toistettavuus ja muihin DevOps-työkaluihin soveltuvuus ovat myös

tutkittavia asioita. Työkalun soveltuvuuden arvioinnin avulla voidaan selvittää, soveltuuko valittu työkalu tutkimusympäristöä vastaavaan todelliseen kehitys- ja tuotantoympäristöön. Työkalun soveltuvuuden arvioinnista kerättävää tietoa voidaan hyödyntää myös valittaessa jatkuvan integraation työkaluja muihin vastaavanlaisiin projekteihin.

4.3 Empiirisen tutkimuksen toteutus

Empiirisen tutkimuksen pääpaino rakentuu tapaustutkimukseen eli jatkuvan integraation implementoimiseen valitun työkalun avulla. Työn empiirisessä osuudessa kerätään tietoja valitun työkalun käyttöönotosta, konfiguroinnista, käytöstä sekä laadullisista ominaisuuksista. Tutkimuksessa jatkuva integraatio implementoidaan valittua päätyökalua hyödyntäen React Native -kehysympäristöllä kehitettäville mobiilisovelluksille todellista ohjelmistotuotantoympäristöä vastaavaan ympäristöön. React Native (2018) on kehysympäristö, jolla voidaan kehittää Android- ja iOS-natiivimobiilisovellukset hyödyntämällä JavaScript-ohjelmointikieltä. Kehysympäristö mahdollistaa tarvittaessa ohjelmiston osien kirjoittamisen natiivisti Androidin Java-ohjelmointikielellä ja iOS-sovelluksen Objective-C- tai Swift-ohjelmointikielillä. Mobiilikehityksen jatkuvan integraation kannalta React Native -kehysympäristöllä kehitettävä mobiilisovellus on otollinen tutkimusta varten, sillä React Native -projektissa voi olla testejä JavaScript-monialustaosalle sekä Android- ja iOS-alustoille. Lisäksi lähdekoodi voidaan paketoida suoritettavaksi sovellukseksi Android- ja iOS-alustoille.

Soveltuvuuden testauksessa tutkimuksen hypoteesi voidaan pohjata valitun työkalun valmistajan listaamiin ominaisuuksiin. Suoritettava tutkimus pyrkii kokeiden avulla varmistamaan ominaisuuksien toiminnallisuuden sekä mahdolliset rajoitteet tutkimuksen React Native -mobiilikehitysympäristössä sekä muiden asetettujen soveltuvuuskriteerien täyttämisen. Työn resurssi- ja aikataulurajoitteiden vuoksi työkalun soveltuvuutta ei voida arvioida useamman kuin yhden mobiilikehitysprojektin kontekstissa. Jatkuvan integraation implementoimisen jälkeen koostetusta raportoinnista tehdään yhteenveto, johon kootaan havainnot työkalun toiminnallisuuksista sekä mahdollisista rajoitteista. Peilaten empiirisessä tutkimuksessa havaittuja asioita aiemmin määriteltyyn vaatimuksiin, voidaan arvioida työkalun soveltuvuutta OCTO3 Oy:n vaatimuksiin. Soveltuvuuden johtopäätös perustuu tapaustutkimuksessa kerättyyn tietoon ja tiedon analysoimiseen.

Tapaustutkimuksen tuloksia voi harvoin yleistää kaikkiin tapauksiin (Kitchenham, Pickard & Pfleeger 1995, s. 53). Tämä on tärkeä huomioitava asia työn empiirisen tutkimuksen tuloksen arvioinnissa, etenkin kun ohjelmistoprojektin koolla on vaikutus siihen, kuinka hyödylliseksi ohjelmistokehitystyökalu osoittautuu (Bruckhaus, Madhavii, Janssen & Henshaw 1996, s. 37). Työssä arvioitava laatu sekä soveltuvuus kohdistuu nimenomaan testattavan projektin kontekstiin, ei suoranaisesti sen ulkopuolelle. Työn empiirisen tutkimuksen tuloksia ja havaintoja sekä niistä johdettuja johtopäätöksiä voidaan hyödyntää soveltuvin osin myös muissa mobiilikehitysprojekteissa. Yleiset havainnot tutkimuksen työkalusta lienee jossain määrin yleistettävissä myös pk-yritys -kontekstin ulkopuolelle, etenkin vastaavilla kehitysteknologioilla kehitettäviin ohjelmistoprojekteihin.

5 TYÖKALUN SOVELTUVUUDEN ARVIOIMINEN

Työn viidennessä kappaleessa esitellään työn empiirinen tutkimus. Kappaleessa esitellään empiirisen tutkimuksen kohteeksi valittu jatkuvan integraation työkalu, kuvataan sitä vasten suoritettavat kokeet ja niiden lopputulokset. Kappaleen lopuksi arvioidaan työkalun soveltuvuutta pk-yrityksen mobiilikehityksen jatkuvaan integraatioon.

5.1 Työkalun esittely

Työn empiiriseen tutkimukseen valikoitiin OCTO3 Oy:n toimesta jatkuvan integraation päätyökalu CircleCI. CircleCI (2018) on jatkuvan integraation ja toimittamisen (*CI ja CD*) suorittamisen mahdollistava kaupallinen työkalu. CircleCI:n ensimmäinen versio julkaistiin vuonna 2016, ja 2017 vuoden syksyllä työkalusta julkaistiin uudistunut toinen versio. Ensimmäisen version tuki lakkautetaan vuoden 2018 jälkimmäisellä puoliskolla. Työkalua käyttäviä organisaatioita on yli 25 000, joihin lukeutuvat organisaatiot kuten Facebook, Kickstarter ja Spotify. CircleCI tukee Bitbucket- ja GitHub-versionhallintajärjestelmiä ja käytettävissä olevia suoritusympäristöjä ovat Docker-kontit ja Linux- ja Mac-virtuaalikoneet. Kaikki ohjelmointikielet, jotka kääntyvät edellä mainituissa suoritusympäristöissä, ovat työkalun tukemia. Työkalua voidaan ajaa organisaation omalla palvelimella tai CircleCI:n pilvipalvelussa. Organisaation omalla palvelimella ajettaessa ainoastaan GitHub-versionhallintajärjestelmän käyttö ja Linux-suoritus ovat mahdollista. Hinnoittelumalliltaan työkalun käyttö perustuu Linux-virtuaalikonttien määrään. Ensimmäinen kontti on ilmainen ja lisäkontit maksavat 50 dollaria kuukaudessa. Avoimen lähdekoodin projekteille tarjotaan neljä konttia ilmaiseksi. MacOS-suorituksen käytön kuukausittaiset kustannukset alkavat 39 dollarista. Konttien hyödyntämistä voi muokata samanaikaisuus (*concurrency*)- ja rinnakkaisuusvalinnoilla (*parallelism*). Työkalun markkinoinnissa käytettäviä muita ominaisuuksia ovat:

- Workflows-toiminnallisuus töiden hallintaan
- Laaja Docker-tuki
- Laitteistoresurssien (CPU/RAM) valitseminen
- Välimuistiominaisuudet
- SSH- ja paikalliset suoritukset
- Tietoturva

- Statistiikka
- Raportointi-integraatiot Jira-, Hipchat- ja Slack-ohjelmistoihin

5.2 Empiirinen tutkimus

Työn empiirinen tutkimus suoritettiin React Native -projektilla, joka generoitiin kehysympäristön virallisia työkaluja käyttäen. React Native -projekti generoitiin siten, että muodostui erilliset Android- ja iOS -natiiviprojektit. Tutkimuksessa testauksen osalta suoritettiin JavaScript-osan sekä Android- ja iOS-projektien yksikkötestit. Tutkimuksessa käytetyn projektin Android-natiivikoodi on Java-ohjelmointikieltä ja iOS-natiivikoodi Objective-C -ohjelmointikieltä. Työn tutkimusympäristön käyttöjärjestelmä oli macOS Sierra 10.12.6. Paikallisessa ympäristössä kehitettiin projektia ja tutkittiin CircleCI-komentorivityökalua. Muuten suoritukset tapahtuivat CircleCI-työkalun palvelinympäristössä. Tutkimuksessa käytettiin CircleCI-työkalun versiota kaksi, sillä ensimmäisen version tuki lakkautetaan lähitulevaisuudessa. Tutkimuksen suorituksen ajanjakso oli noin kahden viikon mittainen.

Taulukko 7. Työn tutkimusympäristö

Ohjelmisto	**Versio**	**Tarkoitus**
CircleCI	2.0	Jatkuvan integraation työkalu
MacOS	Sierra 10.12.6	Paikallisen ympäristön käyttöjärjestelmä
Bitbucket	-	Versionhallintajärjestelmä
GitHub	-	Versionhallintajärjestelmä

CircleCI-työkalun käyttöönotto tapahtuu kirjautumalla palvelun verkkosivustolle joko Bitbucket tai GitHub-käyttäjätunnuksella. Sallittuaan oikeuden käyttää tunnuksen alla olevia projekteja (*repository*), voidaan aloittaa jatkuvan integraation konfigurointi tai suoritus haluttuun projektiin. CircleCI:n integroiminen versionhallintaan oli täysin saumaton. Ainoa havaittu haittapuoli saumattomuudessa on se, että CircleCI:lle annetaan oikeudet kaikkiin tunnuksen alaisiin projekteihin. Tutkimuksessa käytettiin sekä Bitbucket- että GitHub-versionhallintajärjestelmiä. Työn empiirisessä tutkimuksessa suoritettiin taulukon kahdeksan kokeet.

Taulukko 8. Työn empiirisessä tutkimuksessa suoritetut kokeet

Koe	**Kuvaus**
K1	JavaScript-koodin kääntäminen, analysointi ja testien suoritus
K2	Android-koodin kääntäminen, analysointi, testien suoritus ja käyttöönotto
K3	iOS-koodin kääntäminen, analysointi, testien suoritus ja käyttöönotto
K4	Suoritustulosten raportointi

CircleCI-työkalun konfigurointi tapahtuu luomalla projektin juureen ”.circleci”-hakemisto, jonne luodaan YAML-merkintäkielen (YAML Ain’t Markup Language) mukainen ”config.yml”-määrittelytiedosto. Määrittelytiedosto koostuu kolmesta pääotsikosta: versiosta (*version*), töistä (*jobs*) ja työnkuluista (*workflows*). Versio-otsikkoon määritellään käytettävä CircleCI-versio. Työt-otsakkeen alle määritellään työt, jotka suoritetaan jatkuvan integraation rutiinissa. Työnkulku-otsakkeen alle voidaan konfiguroida suoritettavat työt. Työnkulkujen määrittely ei ole välttämätöntä, mutta silloin työt-osiossa tulee olla työ nimeltä ”build”. Työnkuluissa voidaan organisoida suoritettavia töitä asettamalla esimerkiksi suoritusjärjestyksiä, ajastuksia, vaatimuksia manuaaliselle hyväksynnälle ja määrittelemällä versionhallinnan haarat, jotka käynnistävät töiden suoritukset. Työnkulkujen määrittely mahdollistaa töiden suorituksen optimoinnin.

Konfiguraation työt-osioon määritellään suoritettavat työt. Jokaiselle työlle tulee antaa yksilöivä nimi sekä määritellä suoritusympäristö- ja työvaiheet (*steps*). Lisäksi valinnaisia konfiguroitavia asioita ovat komentorivi (*shell*), työhakemisto (*working_directory*), rinnakkaisuus (*parallelism*), ympäristömuuttujat (*environment*), versionhallinnan haarat (*branches*) sekä resurssit (*resource_class*). Suoritusympäristö voi olla Docker-kontti (*docker*), Linux (*machine*)- tai MacOS (*macOS*)-virtuaalikone. Työvaiheisiin määritellään komennot, jotka suoritetaan komentorivillä. Yksittäiselle työvaiheelle voidaan määritellä lisäksi muun muassa ympäristömuuttujia, työhakemisto ja aikakatkaisu. Lähdekoodi voidaan hakea suoraviivaisesti versionhallinnasta SSH-protokollaa hyödyntäen *checkout*-työvaiheessa. Työvaiheisiin voi määritellä vaiheet välimuistiin tallentamiseen (*save_cache*) ja sieltä lataamiseen (*restore_cache*). Lisäksi työhakemiston osia voi halutessaan tallentaa (*persist_to_workspace*) ja ottaa käyttöön (*attach_workspace*). Käyttöönotto (*deploy*) on erityinen suoritusvaihe, joka suoritetaan vain, jos kaikki aikaisemmat suoritusvaiheet ovat suoritettu onnistuneesti. CircleCI:n palvelimille voi tallettaa ajonaikaisia tiedostoja määrittelemällä *store_artifacts*-suoritusvaiheita. Testien metadatan tallennus

store_test_results-työvaiheessa mahdollistaa testitulosten lukemisen suoraan CircleCI-käyttöliittymästä.

Konfiguraation testaaminen on mahdollista käyttäen CircleCI-komentorivityökalua. Komentorivityökalun käyttäminen edellyttää Docker-ohjelmiston asentamista omaan kehitysympäristöön. Komentorivityökalu mahdollistaa YAML-konfigurointitiedoston validoinnin ja töiden suorittamisen paikallisesti. Työkalun käytössä on kuitenkin merkittäviä rajoitteita. Linux-virtuaalikoneen sekä välimuistin käyttö ei ole teknisesti mahdollista. Lisäksi konfiguraatiossa ei voi käyttää ympäristömuuttujia eikä työnkulkuja. Tutkimuksessa havaittiin, että nämä rajoitteet haittasivat komentorivityökalun hyödyntämistä huomattavasti. Komentorivityökalun käyttö edellytti käytännössä toisen konfiguraatiotiedoston luomista, josta ajettiin yksittäisiä töitä konfiguraation testaamiseksi. Tämä menetelmä todettiin hankalaksi ja näin ollen konfiguraation suorituksen testaus tapahtui pääosin tekemällä muutos versionhallintaan ja odottamalla suorituksen lopputulosta CircleCI:n palvelimelta.

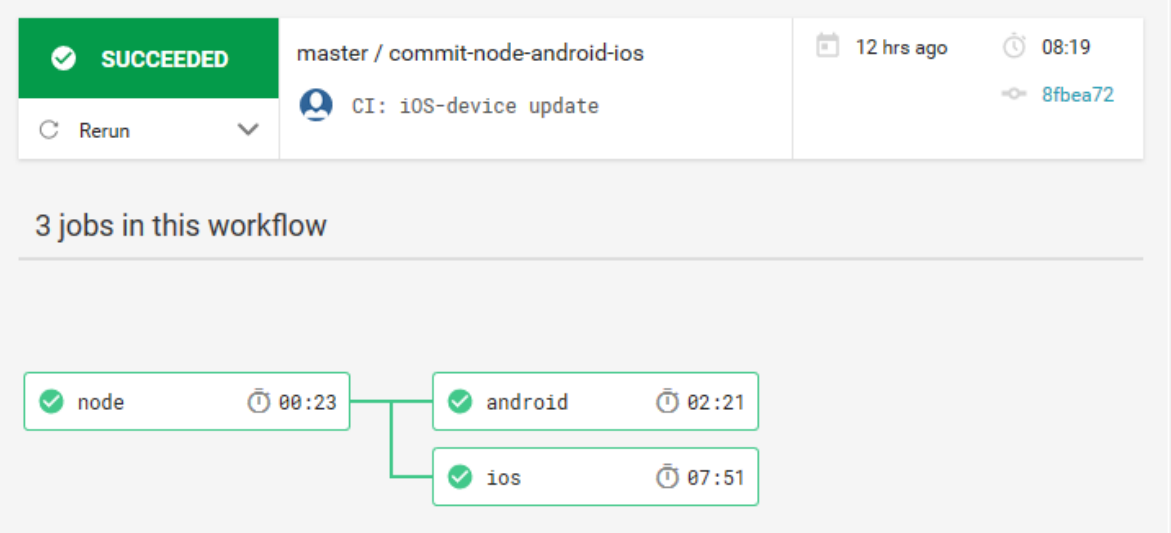
Projektin lopullinen CircleCI-konfiguraatio (Liite 1) koostui kolmesta työstä ja kahdesta työnkulusta. Alustariippumattomalle JavaScript-osan työt suoritettiin Docker-kontin Node.js-levykuvassa. Node.js-projektin riippuvuudet asennettiin yarn-paketinhallintatyökalulla ja suoritusaikaa nopeutettiin konfiguroimalla CircleCI tallentamaan riippuvuudet välimuistiin seuraavia suorituskertoja varten. Projektin testit suoritettiin Jest-työkalulla, jonka lisäksi hyödynnettiin jest-junit-kirjastoa testitulosten kirjoittamiseen CircleCI:n tukemaan XML-formaattiin (Extensible Markup Language). Alustariippumattoman osan riippuvuudet tallennettiin työhakemistoon, jotta Android-projektin suorituksissa samoja vaiheita ei tarvinnut toistaa.

Android-projektin työt suoritettiin Docker-kontissa Android-levykuvassa Gradle-koontiversiointityökalulla. Gradle-työkalun toimiminen edellytti ympäristömuuttujan asettamista konfiguraatioon. Android-työvaiheissa haettiin lähdekoodi versionhallinnasta, otettiin käyttöön JavaScript-osan riippuvuudet sekä ladattiin suoritettavan Android-projektin riippuvuudet. Riippuvuudet tallennettiin CircleCI:n välimuistiin. Yksikkötestien ja analysoinnin suorituksen sekä lopputulosten tallentamisen jälkeen, suoritettiin projektin tuotantoversion koontiversiointi ja muodostuvan .apk-tiedoston (Android Application

Package) tallennus. Tuotantokoontiversion varmentaminen (*code-signing*) onnistui Gradle-työkalua hyödyntämällä. Android-projektin konfiguroinnissa havaittiin, ettei CircleCI 2.0-versio tue simulaattorisuoritusta. Simulaattoria vaativat testit täytyisi suorittaa esimerkiksi Firebase Test Lab -palvelussa.

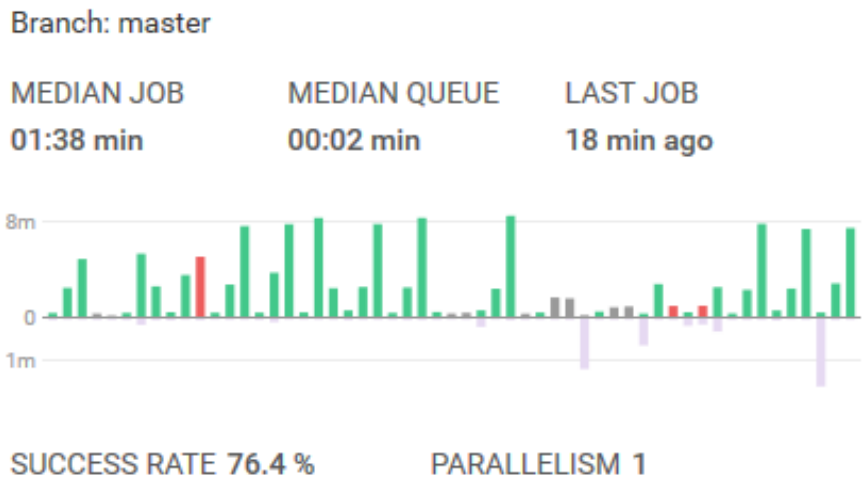
iOS-projektin työt suoritettiin CircleCI:n macOS-ympäristössä Xcodebuild-työkalulla. Mac-ympäristöön tuli asentaa ensin JavaScript-osan riippuvuudet. Koontiversiointi, testaus ja analysointi suoritettiin samassa työvaiheessa ja testitulokset tallennettiin tiedostona CircleCI:n palvelimelle. Sovelluksen tuotantoversion koontiversiointi suoritettiin ilman varmentamista. CircleCI tukee ainoastaan Fastlane Match-työkalua iOS-sovellusten lähdekoodin varmentamiseen, joten Fastlane-työkalujen hyödyntäminen on edellytys oikealla laitteella suoritettavan tai sovelluskauppaan lähetettävän koontiversion luomiselle. CircleCI tukee Android-ympäristöstä poiketen testien suorittamista simulaattorissa MacOS-ympäristössä.

Työnkuluissa määriteltiin kaksi eri töiden suoritusrutiinia. Toinen työnkulku suoritetaan jokaisesta päälinjan koodimuutoksesta ja toinen ajastetusti kerran vuorokaudessa. Työnkulkujen määrittely havaittiin tutkimuksessa hyväksi ominaisuudeksi. Tutkimuksessa käytetyn projektin React Native -alustariippumattoman JavaScript-osan työt suoritettiin ensin, jonka jälkeen Android- ja iOS-työt suoritettiin rinnakkain (kuva 1). Täten voitiin hyödyntää Android-projektissa JavaScript-osan riippuvuuksia ilman uudelleenlatauksia sekä jatkettiin suoritusta vain alustariippumattoman osan rutiinien onnistuessa. Suuremman kokoluokan projekteissa ominaisuuden arvo vain kasvaa. Työnkulkujen lisäksi CircleCI:tä voi operoida rajapinnan kautta. Rajapinnan kautta voi muun muassa hakea tietoa suorituksista, suorituksen aikana tallennettuja tiedostoja sekä käynnistää suorituksia. Jatkuvan integraation kokonaissuoritusaika tutkimuksen React Native -projektissa oli keskimäärin noin kahdeksan minuuttia. Alustariippumattoman osan töiden suorittaminen kesti keskimäärin 25 sekuntia, Android-töiden kaksi ja puoli minuuttia ja iOS-töiden vajaa kahdeksan minuuttia. iOS-töiden suoritusaikaan vaikutti merkittävästi valitun simulaattorilaitteen tyyppi.



Kuva 1. Projektin työnkulku -näkymä

CircleCI-työkalun konfiguroiminen tutkimuksen React Native -projektiympäristössä todettiin kokonaisuudessaan melko yksinkertaiseksi, joskin aikaa vieväksi. CircleCI:n dokumentaatiosta ja viralliselta foorumilta löytyi lähes kaikki tarvittava tieto jatkuvan integraation implementoimiseen tutkimuksen projektiin. CircleCI:n verkkosivuston käyttöliittymän käytettävyys todettiin tutkimusjakson aikana hyväksi. Käyttöliittymästä voi säätää yleisiä ja projektikohtaisia asetuksia, lukea suoritettujen töiden lokitietoja, ladata suorituksen aikana tallennettuja tiedostoja sekä tarkastella tilastoja suorituksista (kuva 2). CircleCI lähettää oletusasetuksilla tiedon jatkuvan integraation onnistumisesta suoraan versionhallintaan. Tutkimuksen aikana huomattiin, ettei CircleCI tue käyttötapausta, jossa jatkuvan integraation rutiinit suoritetaan päälinjaan ja muihin haaroihin vain silloin, jos niistä oli lähetetty pyyntö *pull request* -toiminnallisuudelle. Tutkimuksessa työkalu konfiguroitiin raportoimaan sähköpostin välityksellä.



Kuva 2. Projektin suoritustilastot

5.3 Työkalun soveltuvuus pk-yritykseen

Työkalun soveltuvuutta arvioitiin peilaamalla empiirisessä tutkimuksessa saavutetun jatkuvan integraation toteutusta kirjallisuuskatsauksesta kerättyyn aineistoon. Tutkimuksessa todettiin CircleCI-työkalun mahdollistavan jatkuvaan integraatioon kuuluvat prosessit tutkimuksen mobiilikehitysprojektissa. Mobiilialustoille kirjoitetun lähdekoodin kääntäminen, testaaminen, analysointi sekä käyttöönotto todettiin mahdolliseksi testattavalla työkalulla. Tietokantaintegraation osalta ei voida ottaa kantaa tämän työn kontekstissa, sillä empiirisen tutkimuksen testiprojekti ei vaatinut erillistä tietokantaintegraatiota.

Toisena soveltuvuuden tarkastelun viitekehyksenä käytettiin tutkimuksessa ohjelmistotyökalun laatua. Laadun arvioinnin viitekehyksenä hyödynnettiin ISO/IEC 25010-standardia. Työkalun toiminnallinen sopivuus mobiilikehityksen jatkuvaan integraatioon todettiin tutkimuksessa riittäväksi muutamin rajoittein. Näistä merkittävin on suoran tuen puuttuminen Android-simulaattorin käytölle. Työkalun tehokkuus oli tutkimuksen React Native -kehysympäristöllä toteutettavan ohjelmistoprojektin kontekstissa kelvollinen. Töiden suoritusajat olivat, tutkimuksen projektiympäristö huomioon ottaen, hyväksyttäviä. Yhteensopivuus muiden järjestelmien kanssa oli kohtuullinen. Versionhallintaintegraatio vain GitHub- ja Bitbucket-järjestelmien kanssa rajoittaa yhteensopivuutta. Toisaalta CircleCI-työkalun avoimet kontti- ja virtuaalikoneympäristöt mahdollistavat laajan yhteensopivuuden lukuisten muiden työkalujen kanssa.

Käytettävyydessä ei havaittu merkittäviä puutteita. Työkalun käyttöliittymä tuki tehtäviä toimenpiteitä ja työkalun konfigurointi todettiin melko yksinkertaiseksi dokumentaation ja muun tuen avulla. Käyttöliittymässä näytettävät virheilmoitukset olivat selkeitä ja niiden avulla esimerkiksi konfigurointivirheet voitiin paikallistaa sekä korjata. Työkalun luotettavuus todettiin hyväksi. Empiirisen tutkimuksen testijakson aikana ei törmätty yhteenkään työkalusta johtuvaan virheeseen tai ongelmaan. Turvallisuuden näkökulmasta työkalua ei juurikaan päässyt työn empiirisessä tutkimuksessa arvioimaan. Tutkimuksessa kuitenkin havaittiin, että työkalun käyttöönotto edellytti oikeuden antamista kaikkiin versionhallinnan projekteihin. Jatkuva integraatio suoritettiin CircleCI-työkalun palvelimilla, joten turvallisuus oli työkalun palvelinkonfiguraation armoilla. Ylläpidettävyyden osalta työkalu on hyvällä tasolla. Toimivan konfiguraation määriteltyään, työkalun käyttö ei edellyttänyt muita ylläpitotoimenpiteitä. Paikallisten komentorivityökalujen käyttö todettiin ylläpitoa helpottavaksi tekijäksi, etenkin konfiguraation validoinnissa. Työkalun siirrettävyyden näkökulmasta työkalua on hankala arvioida, sillä tutkimuksessa työkalua käytettiin CircleCI:n palvelinympäristössä. Tuki työkalun käyttöön omassa palvelinkonfiguraatiossa on siirrettävyyttä edistävä tekijä. Useat jatkuvan integraation työkalut käyttävät konfiguroinnissa YAML-tiedostoja, joten CircleCI-konfiguraation perusajatusta voi hyödyntää pohjana toisiin jatkuvan integraation työkaluihin, vaikka syntakseissa onkin eroja.

Kolmantena soveltuvuuden kriteerinä olivat OCTO3 Oy:n asettamat vaatimukset. Työkalun tuli mahdollistaa React Native -mobiilikehitysprojektin jatkuva integraatio. Empiirisen tutkimuksen kohteena oli React Native -projekti, jonka kontekstissa työkalu todettiin soveltuvaksi jatkuvan integraation toteuttamiseen. Työkalun todettiin myös tukevan muita DevOps-menetelmiä tarjoamalla suoran alustan jatkuvan toimittamisen implementoimiseen samalla konfiguraatiotiedostolla. Empiirisessä tutkimuksessa implementoitu jatkuvan integraation ympäristö on kohtuullisen hyvin toistettavissa myös toisiin mobiiliprojekteihin. Aikaisempia konfiguraatioita voidaan hyödyntää lähtökohtana muiden vastaavanlaisien mobiilikehitysprojektien jatkuvan integraation implementoinnissa.

Yleisiä havaittuja työkalun soveltuvuutta rajoittavia tekijöitä ovat työkalun hinnoittelu, komentorivityökalun vajavaisuus, tuettujen versionhallintajärjestelmien rajallisuus, vaadittu sitoutuminen varmennetussa iOS-koontiversioinnissa Fastlane Match -työkaluun sekä

Android-simulaattorituen puuttuminen. Mac-ympäristön tuen puuttuminen organisaation omalla palvelimella ajettaessa on myös merkittävä puute mobiilikehityskontekstissa. CircleCI on työkaluna melko tuore ja näin ollen työkalu kehittyy jatkuvasti. Jatkuvat muutokset ja mahdollinen epävakaus voidaan nähdä työkalun soveltuvuutta rajoittavana tekijänä.

6 JOHTOPÄÄTÖKSET

Tämä kappale on omistettu työn tutkimuskysymykseen vastaamiseen sekä toteutetun tutkimuksen reflektoimiseen. Lisäksi kappaleessa kuvataan työn aiheeseen liittyvät jatkotutkimuskohteet.

6.1 Päätelmä

Kirjallisuuskatsaukseen sekä empiiriseen tutkimukseen pohjautuen jatkuvan integraation optimoimiseen ei ole oikotietä. Jokaisen ohjelmistokehitysprojektin ollessa hieman erilainen, ei jatkuvaan integraatioon ole yleistä sapluunaa, jota noudattamalla pääsisi parhaaseen lopputulokseen. Pääosa jatkuvan integraation työkaluista soveltuvat hyvin tarkoitukseensa ja ne eivät vaadi valtavia ylläpitopanostuksia. Jatkuvan integraation optimoiminen on kuitenkin merkittävästi kiinni sitä suorittavasta tahosta, ohjelmistokehittäjistä. Optimointi voi edellyttää uusien toimintatapojen omaksumista sekä asennoitumista jatkuvan integraatioon työnkulkua edistävällä tavalla. Toimintatavoista merkittävin on testauksen kehittäminen, sillä jatkuva integraatio hyötyy merkittävästi laadukkaasta ja automatisoidusta testausrutiinista. Laadukas ja lyhytkestoinen testirutiini paljastaa suuremmalla todennäköisyydellä ohjelmiston ongelmakohdat ja virheet mahdollisimman aikaisessa vaiheessa sekä optimoi jatkuvan integraation työnkulkua. Jatkuvan integraation optimoinnissa isossa roolissa on myös kehittäjien osaaminen. Aikaisemmat kokemukset jatkuvan integraation implementoinnista vähentävät implementointiin kuluvaa aikaa sekä mahdollistavat oikeiden työkalujen valinnan ja työnkulun kehittämisen. Aikaisemmin toimivaksi todettu implementaatio on hyvä lähtökohta jatkuvan integraation hyödyntämiselle uudessa vastaavanlaisessa projektissa.

Työhön valittu pk-yritys -konteksti ei sinällään tuo merkittävää muutosta näkemykseen jatkuvan integraation optimoinnista. Jatkuvan integraatioon implementoimiseen sekä ylläpitoon kuluvat resurssit yrityksen alkutaipaleella voivat olla merkittävät, mutta pitkällä aikavälillä jatkuvan integraation tuottama lisäarvo kattaa varmasti siihen kuluvat resurssit. Ohjelmiston kehityksen sujuvuuden sekä sen laadunvalvonnan parantuminen ovat pääasioita, joita jatkuva integraatio voi tarjota. Lisäksi jatkuva integraatio toimii ohjelmistokehittäjälle hyvänä ponnistusalustana oman ammattitaidon kehittämiselle, sillä jatkuvan integraation optimointi edellyttää laadukkaiden automaattitestien kirjoittamista.

Täytyy tietenkin huomioida, ettei kaikissa pk-yrityksen pienemmissä kehitysprojekteissa välttämättä ole sijaa mittavalle jatkuvan integraation implementoimiselle. Mobiiliohjelmistokehitys tuo jatkuvan integraation implementoimiseen haasteita laitteiden monimuotoisuudella sekä testauksen hankaluudella ja näin ollen jatkuvan integraation työnkulun optimointi voi olla haastavampaa. Työn tutkimuksessa havaittiin, ettei kaikki suosituimmat jatkuvan integraation työkalut välttämättä tarjoa tukea mobiilikehitykselle. Natiivi- ja alustariippumattoman mobiiliohjelmistokehityksen erot voivat myös aiheuttaa haasteita jatkuvan integraation implementoimiseen. Työn empiirisessä tutkimuksessa havaittiin, että etenkin mobiilikehityskontekstissa jatkuvan integraation optimaalisuus riippuu paljon käytettävistä työkaluista. Tutkimuksen pohjalta voidaan todeta, että jatkuvan integraation optimointi mobiilikehityskontekstissa edellyttää suuremmissa määrin oikeiden työkalujen valintaa ja niiden tuntemista.

Työn lisätutkimuskysymyksinä oli selvittää, soveltuuko valittu työkalu mobiilikehityksen tarpeisiin ja tukeeko valittu työkalu muita DevOps-käytänteitä. Työn empiirisessä tutkimuksessa implementoitiin jatkuva integraatio React Native -kehysympäristöllä kehitettyyn ohjelmistoprojektiin CircleCI-työkalulla. Tutkimuksessa hyödynnetty projekti sisälsi alustariippumatonta- ja Android- ja iOS-natiivikoodia. Lähdekoodin koontiversiointi suoritettiin Android- ja iOS-alustoille natiivityökaluilla. Työn empiirisessä tutkimuksessa todettiin CircleCI-työkalun soveltuvan natiivi- ja React Native -kehysympäristöllä toteutettavaan mobiilikehitykseen. Empiirisessä tutkimuksessa todettiin myös työkalun valmius jatkuvan toimittamisen toimenpiteiden suoraviivaiseen implementointiin ja tarjoavan tehokkaan sekä luotettavan alustan ohjelmistokehitysketjun tehostamiseen.

6.2 Reflektio

Työn tutkimus eteni suunnitelman mukaisesti aikataulutusta lukuun ottamatta. Työn tutkimuskysymyksiin pystyttiin vastaamaan, vaikkei mobiilikehityksen jatkuvasta integraatiosta merkittävästi aikaisempaa tutkimustietoa löytynytkään. Jatkuva integraatio on yleisenä toimintamallina kuitenkin melko yksiselitteinen ja yleistettävissä ohjelmistokehitysprojektityyppien rajojen yli. Kaikki ohjelmistoprojektit sisältävät lähdekoodia, joka tulisi saattaa suoritettavaan muotoon ja sen laatu tulisi varmistaa. Näin ollen mobiilikehitys työn viitekehyksenä ei vaikuttanut tutkimukseen kovinkaan merkittävästi.

Työn empiiriseen osuuteen, jatkuvan integraation työkalun soveltuvuuden koestamiseen, kului odotettua enemmän aikaa. Empiiriselle tutkimukselle olisi voinut varata enemmän aikaa ja tutkimuksen suunnitelma olisi voinut olla tarkempi. Empiirisessä tutkimuksessa saavutettiin kuitenkin asetetut tutkimustavoitteet. Työkalun soveltuvuutta olisi voinut arvioida tarkemmin, jos empiiriseen tutkimukseen olisi hyödynnetty projektia, jossa olisi ollut laajempi testikattaus. Työkalun soveltuvuus laajempaan automatisoituun integraatio- ja systeemitestaukseen jäi työn tutkimuksessa selvittämättä. Työkalun tuotannollinen käyttö laajemmassa ohjelmistokehitysprojektissa ei myöskään ollut työn olosuhteissa mahdollista. Tuotannollisessa käytössä työkalusta olisi voinut kerätä enemmän dataa ja empiiristä tutkimusta olisi voinut lähestyä enemmän työn päätutkimuskysymyksen kautta.

6.3 Jatkotutkimus

Työn päätutkimuskysymykseen vastattiin pääosin kirjallisuuskatsauksen kautta. Mobiilikehityksen jatkuvan integraation optimointia voisi tutkia laajemmin todellisten tuotannollisten ohjelmistoprojektien kautta. Useamman mobiilikehitysprojektin vertailu niissä hyödynnettävien jatkuvan integraation implementaatioiden osalta tarjoaisi laajemman kuvan siitä, millaiset toimintamallit ja työkalut ovat optimaalisimpia mobiilikehitysprojekteihin. Työn empiiristä tutkimusta voisi laajentaa vertailemalla useampaa jatkuvan integraation työkalua. Markkinoilla on useita työkaluja, jotka markkinoivat tarjoavansa tuen mobiilikehitykselle. Olisi hyvä selvittää, onko työkaluissa merkittäviä eroja. Jatkotutkimuskohteena voisi olla myös mobiilikehityksen jatkuva toimittaminen, jota työssä sivuttiin.

LÄHTEET

Agile Alliance 2018. 12 Principles Behind the Agile Manifesto [verkkodokumentti]. [Viitattu 24.3.2018]. Saatavilla https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/

Aho, A. V., Lam, S. M., Sethi, R. & Ullman J. D. 2006. Compilers: Principles, Techiniques and Tools, 2nd edition, Addison Wesley, Boston.

Akour, M., Falah, B. Ahmad, A. Bouriat, S. & Alemerien, K. 2016. Mobile Software Testing: Thoughts, Strategies, Challenges and Experimental Study. *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 6, s.12-19

All About Circuits 2018. What are Integrated Development Environments? [verkkodokumentti]. [Viitattu 21.05.2018]. Saatavilla https://www.allaboutcircuits.com/technical-articles/what-are-integrated-development-environments/

Android 2018. Developers: Android Studio [verkkodokumentti]. [Viitattu 21.05.2018]. Saatavilla https://developer.android.com/studio/

Apple 2018. Developer Technical Note: Building from Command Line with Xcode FAQ [verkkodokumentti]. [Viitattu 21.05.2018] Saatavilla https://developer.apple.com/library/archive/technotes/tn2339/_index.html

Astegic 2018. 6 Mobile Test Automation Tools You Should Know About [verkkodokumentti]. [Viitattu 21.05.2018]. Saatavilla https://www.astegic.com/6-mobile-test-automation-tools-know/

Atlassian 2018a. DevOps: Breaking the Development-Operations barrier [verkkodokumentti]. [Viitattu 27.3.2018]. Saatavilla https://atlassian.com/devops

Atlassian 2018b. Bamboo [verkkodokumentti]. [Viitattu 6.4.2018]. Saatavilla https://fi.atlassian.com/software/bamboo

Atlassian 2018c. What is version control [verkkodokumentti]. [Viitattu 6.4.2018]. Saatavilla https://www.atlassian.com/git/tutorials/what-is-version-control

Bitrise 2018. Bitrise DevCenter – What is Bitrise? [verkkodokumentti]. [Viitattu 6.4.2018]. Saatavilla https://devcenter.bitrise.io/

Bosch, J. 2014. Continuous Software Engineering, Springer International Publishing, Switzerland

Bruckhaus, T., Madhavii, N. H., Janssen, I. & Henshaw, J. 1996. The impact of tools on software productivity. *IEEE Software*, vol. 13, no. 5, s. 29-38

CircleCI 2018. CircleCI [verkkosivusto]. [Viitattu 5.4.2018]. Saatavilla https://circleci.com/

Duvall, P. M., Matyas, S. & Glover, A. 2007. Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley Professional

Ebert, C., Gallardo, G., Hernantes, J. & Serrano, N. 2016. DevOps. *IEEE Software*, vol. 33, no. 3, s. 94-100.

Firth, R., Mosley, V., Pethia, R., Roberts, L. & Wood, W. 1987. A Guide to the Classification and Assessment of Software Engineering Tools.

Fowler, M. 2006. Continuous Integration [verkkodokumentti]. [Viitattu 25.3.2018]. Saatavilla https://www.martinfowler.com/articles/continuousIntegration.html

Gimpel, J. 2014. Software That Checks Software: The Impact of PC-lint. *IEEE Software*, vol. 31, no. 1, s. 15-19

GitLab CI 2018. Languages, frameworks, OSs [verkkodokumentti]. [Viitattu 5.4.2018]. Saatavilla https://docs.gitlab.com/ee/ci/examples/#languages-frameworks-oss

G2Crowd 2018. Continuous Integration [verkkodokumentti]. [Viitattu 5.4.2018]. Saatavilla https://www.g2crowd.com/categories/continuous-integration

Hilton, M., Tunnell, T., Huang, K., Marinov, D. & Dig, D. 2016. Usage, costs and benefits of continuous integration in open-source projects. Automated Software Engineering (ASE), s. 426-437

Humble, J. & Farley, D. 2010. Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation, Addison-Wesley Professional

InfoQ 2018. The Continuous Delivery Maturity Model [verkkodokumentti]. [Viitattu 31.5.2018]. Saatavilla https://www.infoq.com/articles/Continuous-Delivery-Maturity-Model

ISO/IEC 2011. ISO/IEC 25010 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, Switzerland

Jenkins 2018. Plugins [verkkosivusto]. [Viitattu 5.4.2018]. Saatavilla https://plugins.jenkins.io/

Jobe, W. 2013. Native Apps Vs. Mobile Apps. *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 7, no. 4, s. 27-32

Joorabchi, M. E., Mesbah, A. & Kruchten, P. 2013. Real Challenges in Mobile App Development. Empirical Software Engineering and Measurement, ACM/IEE International Symposium on Empirical Software Engineering and Measurement, s. 15-24

Kitchenham, B., Pickard, L. & Pfleeger, S.L. 1995. Case studies for method and tool evaluation. *IEEE Software*, vol. 12, no. 4, s. 52-62

Kumar, D. & Mishra, K. 2016. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science*, vol. 79, s. 8-15

Lianping, C. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, vol. 32, no. 2, s. 50-54

Meyer, M. 2014. Continuous Integration and Its Tools. *IEEE Software*, vol. 31, no. 3, s. 14-16

NetMarketShare 2018. Operating System Market Share. [verkkodokumentti]. [Viitattu 27.5.2018]. Saatavilla https://netmarketshare.com/operating-system-market-share

Nevercode 2018. [verkkosivusto]. [Viitattu 5.4.2018]. Saatavilla https://nevercode.io/

OCTO3 2017. DevOps jatkuvan kehittämisen tukena [verkkodokumentti]. [Viitattu 7.4.2018]. Saatavilla http://www.octo3.fi/devops-jatkuvan-kehittamisen-tukena/

Polo, M., Reales, P., Piattini, M. & Ebert, C. 2013. Test Automation. *IEEE Software*, vol. 30, no. 1, s. 84-89

Riungu-Kalliosaari, L., Mäkinen, S., Lwakatare, L. E., Tiihonen, J. & Männistö, J. 2016. DevOps Adoption Benefits and Challenges in Practice: A Case Study. 17th International Conference on Product-Focused Software Process Improvement, November 22-24, Trondheim, Norway s. 590-597

Roche, J. 2013. Adopting DevOps Practices in Quality Assurance. *Communications of The Acm*, vol. 56, no. 11, s. 38-43

React Native 2018. Building native mobile apps with JavaScript and React [verkkodokumentti]. [Viitattu 15.5.2018]. Saatavilla https://facebook.github.io/react-native/

Shahin, M., Ali Babar, M. & Zhu, L. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, vol. 5, s. 3909-3943

SonarQube 2018. DevOps Integration [verkkodokumentti]. [Viitattu 6.4.2018]. Saatavilla https://www.sonarqube.org/features/integration/

Sqreen 2018. How To Choose Your CI Tool [verkkodokumentti]. [Viitattu 28.3.2018]. Saatavilla https://blog.sqreen.io/how-to-choose-your-ci-tool

Ståhl, D. & Bosch, J. 2013. Experienced benefits of continuous integration in industry software product development: A case study. 12th IASTED International Conference on Software Engineering, February 11 – 13, Innsbruck, Austria

Travis 2018. Languages [verkkodokumentti]. [Viitattu 5.4.2018]. Saatavilla https://docs.travis-ci.com/user/languages/

UrbanCode 2018. Continuous Delivery Maturity Model [verkkodokumentti]. [Viitattu 31.5.2018]. Saatavilla https://developer.ibm.com/urbancode/docs/continuous-delivery-maturity-model/

Virmani, M. 2015. Understanding DevOps & Bridging the gap from Continuous Integration to Continuous Delivery. Fifth International Conference on the Innovative Computing Technology (INTECH), s. 78-82

Wang, M., Meng, Y., Han, L. & Bai H. 2014. From Source Code Analysis to Static Software Testing. IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), s. 1280-1283

Wassermann, A. 2010. Software engineering issues for mobile application development. Proceedings of the FSE/SDP workshop on Future of software engineering research, November 7-8, New Mexico, USA s. 397-400

Zhu, L., Bass, L. & Champlin-Scharff, G. 2016. DevOps and Its Practices. *IEEE Software*, vol. 33, no. 3, s. 32-34

LIITE 1. Empiirisen tutkimuksen CircleCI YAML-konfigurointitiedosto

```yaml
version: 2

jobs:
  node:
    working_directory: ~/reactnativeci
    docker:
      - image: circleci/node:8
    steps:
      - checkout
      - restore_cache:
          key: yarn-v1-{{ checksum "yarn.lock" }}-{{ arch }}
      - restore_cache:
          key: node-v1-{{ checksum "package.json" }}-{{ arch }}
      - run:
          name: Install dependencies
          command: yarn install
      - save_cache:
          key: yarn-v1-{{ checksum "yarn.lock" }}-{{ arch }}
          paths:
            - ~/.cache/yarn
      - save_cache:
          key: node-v1-{{ checksum "package.json" }}-{{ arch }}
          paths:
            - node_modules
      - run:
          name: Test
          # Runs Jest-tests with jest-junit plugin to output results as XML.
          # Command:
          # JEST_JUNIT_OUTPUT="test-results/jest/jest-results.xml" jest --ci
          # --testResultsProcessor="jest-junit"
          command:
            yarn run test-ci
      - run:
          name: Inspect
          command:
            yarn run lint
      - persist_to_workspace:
          root: ~/reactnativeci
          paths:
            - node_modules
      - store_test_results:
          path: test-results
      - store_artifacts:
          path: test-results/jest
          destination: test-results/jest
      - store_artifacts:
          path: lint-results
          destination: test-results/
```

(jatkuu)

Liite 1. (jatkoa)

```
android:
  working_directory: ~/reactnativeci/android
  docker:
    - image: circleci/android:api-27-node8-alpha
  environment:
      # Gradle requires TERM environment variable in CircleCI
      TERM: dumb
  steps:
    - checkout:
        path: ~/reactnativeci
    - attach_workspace:
        at: ~/reactnativeci
    - restore_cache:
        key: jars-{{ checksum "build.gradle" }}-{{ checksum  "app/build.gradle" }}
    - run:
        name: Install dependencies
        command:
          ./gradlew androidDependencies
    - save_cache:
        paths:
          - ~/.gradle
        key: jars-{{ checksum "build.gradle" }}-{{ checksum  "app/build.gradle" }}
    - run:
        name: Build, test and inspect
        command:
            ./gradlew lint test
    - store_test_results:
        path: app/build/test-results
    - run:
        name: Build release
        command: ./gradlew clean assembleRelease --no-daemon
    - store_artifacts:
        path: app/build/reports
        destination: reports/
    - store_artifacts:
        path: app/build/outputs/
        destination: outputs/
```

(jatkuu)

Liite 1. (jatkoa)

```
ios:
  macos:
    xcode: "9.4.0"
  working_directory: ~/reactnativeci

  steps:
    - checkout
    - restore_cache:
        key: yarn-v1-{{ checksum "yarn.lock" }}-{{ arch }}
    - restore_cache:
        key: node-v1-{{ checksum "package.json" }}-{{ arch }}
    - run:
        name: Install dependencies
        command: yarn install
    - save_cache:
        key: yarn-v1-{{ checksum "yarn.lock" }}-{{ arch }}
        paths:
          - ~/.cache/yarn
    - save_cache:
        key: node-v1-{{ checksum "package.json" }}-{{ arch }}
        paths:
          - node_modules
    - run:
        name: Build, test and inspect
        command: xcodebuild
                  -sdk iphonesimulator
                  -destination 'platform=iOS Simulator,OS=11.2,name=iPhone X'
                  -scheme "reactnativeci"
                  clean build test analyze |
                  xcpretty --color -r junit -o test-results/results.xml
        working_directory: ios
    - run:
        name: Build release
        command: xcodebuild
                  -scheme "reactnativeci"
                  -sdk iphoneos
                  -configuration release
                  CODE_SIGN_IDENTITY=""
                  CODE_SIGNING_REQUIRED=NO | xcpretty
        working_directory: ios
    - store_test_results:
        path: ios/test-results
    - store_artifacts:
        path: ios/test-results
        destination: ios-test-results
```

(jatkuu)

Liite 1. (jatkoa)

```
workflows:
  version: 2
  commit-node-android-ios:
    jobs:
      - node:
          filters:
            branches:
              only: master
      - android:
          filters:
            branches:
              only: master
          requires:
            - node
      - ios:
          filters:
            branches:
              only: master
          requires:
            - node
  nightly-node-android-ios:
    triggers:
      - schedule:
          cron: "0 0 * * *" # UTC midnight
          filters:
            branches:
              only:
                - master
    jobs:
      - node:
          filters:
            branches:
              only: master
      - android:
          filters:
            branches:
              only: master
          requires:
            - node
      - ios:
          filters:
            branches:
              only: master
          requires:
            - node
```