hyväksymispäivä arvosana

arvostelija

Jatkuva integraatio

Piia Hartikka

Helsinki 12.12.2017

Kandidaatintutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Piia Hartikka

Työn nimi — Arbetets titel — Title

Jatkuva integraatio

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	12.12.2017	19 sivua

Tiivistelmä — Referat — Abstract

Jatkuva integraatio on työskentelytapa, jossa ohjelmistokehittäjät integroivat työnsä ohjelmiston pääversioon vähintään kerran päivässä. Sen tarkoituksena on toteuttaa ohjelmistotuotannon riskialttiit integraatiot mahdollisimman usein, jotta virheet huomataan ja korjataan ennen kuin ne kasvavat suuriksi. Näin ohjelmistotuotannon projektit muuttuvat ennustettavammiksi. Integraatioon kuuluu ohjelmiston kokoaminen erillisellä jatkuvan integraation palvelimella. Kokoamisen yhteydessä suoritetaan regressiotestaaminen, joka keskittyy testaamaan muutoksen vaikutusta ohjelmistoon ja sen oikeellisuuteen.

ACM Computing Classification System (CCS): Software and its engineering → Software creation and management → Software verification and validation

Avainsanat — Nyckelord — Keywords
jatkuva integraatio, regressiotestaaminen

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto **1**

2 Jatkuva integraatio **3**

2.1 Koontiversio 5

2.2 Jatkuvan integraation hyvät ja huonot puolet 6

2.3 Case: Jatkuva integraatio Googlella 9

3 Regressiotestaaminen jatkuvassa integraatiossa **12**

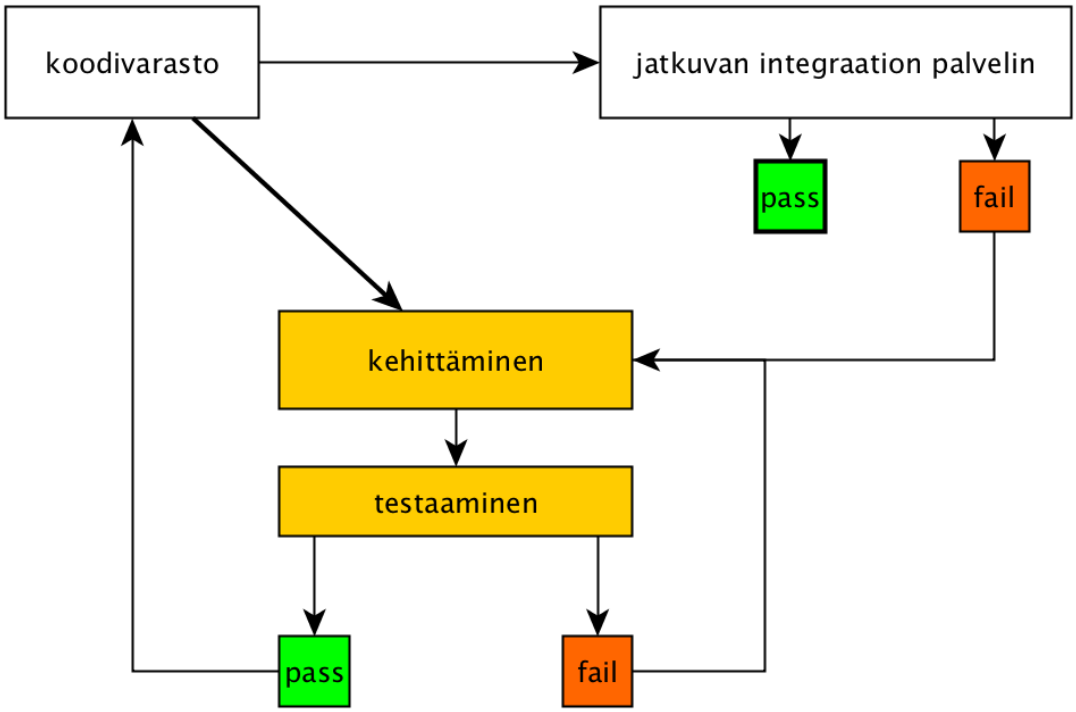
3.1 Kustannukset 13

4 Yhteenveto **16**

Lähteet **18**

1 Johdanto

Jatkuva integraatio korvaa perinteisen ohjelmistokehityksen päättävän integraatiovaiheen. Se on työskentelytapa, jossa ohjelmistokehittäjä integroi työtään vähintään kerran päivässä ohjelmistokehityksen päälinjaan. Ohjelmistokehittäjä aloittaa työskentelynsä hakemalla ohjelmiston uusimman version yhteisestä koodivarastosta ja kommitoi tehdyn ja testatun työn uusimmaksi versioksi saman päivän aikana. Integraatiossa ohjelmisto kootaan ja ohjelmakoodin sisältämät automatisoidut testit ajetaan jatkuvaan integraatioon varatulla palvelimella. Tämä on esitetty myös kuvassa 1. Jatkuvassa integraatiossa jokainen kehityssykli loppuu vasta kuin jatkuvan integraation palvelin ilmoittaa kommitin koontiversion (build) onnistumisesta [Fow06].



Kuva 1: Jatkuva integraatio käytännössä.

Jatkuvan integraation testitoiminnot voidaan jakaa karkeasti kahteen luokkaan: tes-

taaminen ennen kommitointia ja kommitoinnin jälkeen. Kommitoinnin jälkeen suoritettavalla testaamisella tarkoitetaan testaamista ohjelmiston koonnin (build) yhteydessä jatkuvan integraation palvelimella. Projektista riippuen kommitointia edeltävä testaaminen voi tapahtua ohjelmistokehittäjän työkoneella tai erillisellä testipalvelimella [ERP14]. Ohjelmistokehittäjän on testattava työnsä ennen integraatiota välttääkseen turhaan rikkomasta ohjelmiston pääversiota, sillä koko ohjelmistokehitystiimi työskentelee sen parissa [MSB17].

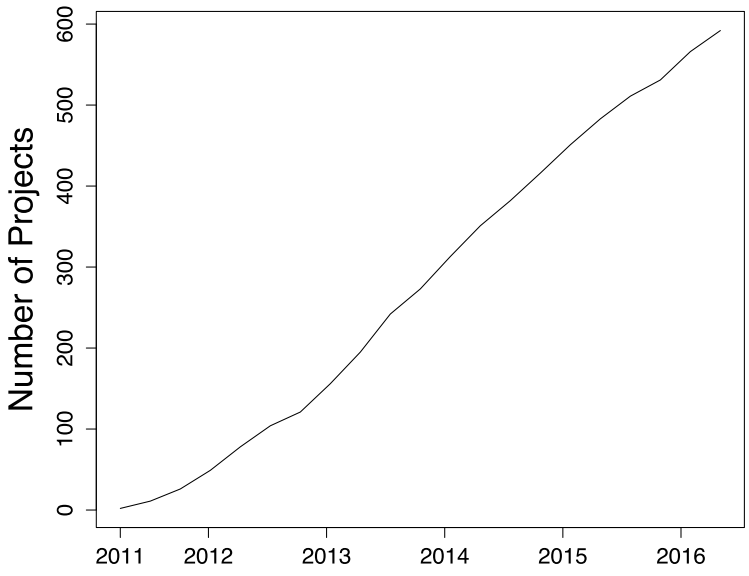
Regressiotestaamisella varmistetaan ohjelmiston oikeellisuus muutoksen jälkeen. Regressiotestaaminen voidaan toteuttaa esimerkiksi valitsemalla kokemuksen perusteella sopiva joukko yksikkö-, integraatio- ja järjestelmätestejä. Kaikkien mahdollisten testien suorittaminen ei ole kannattavaa, sillä se kuluttaa palvelinresursseja ja viivyttää ohjelmiston tilasta kertovan palautteen saapumista. Ohjelmiston kokoamisen ja testien ajamisen kun pitäisi suosituksen mukaan kestää alle kymmenen minuuttia. Testikokonaisuutta voi rajata priorisoimalla testitapauksia tai valitsemalla informatiivisimmat testit koodipohjan (codebase) analysoinnin perusteella. Regressiotestejä kehitetään ajan kuluessa varsinaisen ohjelmakoodin rinnalla ja siihen voi kulua yhtä paljon työaikaa kuin varsinaiseen ohjelmistokehitykseen [LIH17].

Jatkuva integraatio on ohjelmistotuotannossa suhteellisen nuori menetelmä. Historiallisesti sen esitteli ensimmäisen kerran Grady Booch vuonna 1991, mutta vasta 1990-luvun lopulla se otettiin osaksi Extreme Programming -metodologiaa. Jatkuva integraatio levisi ja sai laajaa kannatusta Martin Fowlerin kirjoitettua siitä blogitekstin vuonna 2000 ja julkaistua myöhemmin myös kirjan [SB13]. Vielä vuonna 2016 jatkuvan integraation tutkimusta on pidetty vähäisenä [HTH+16].

Tämän kandidaatintutkielman luvussa yksi käsitellään jatkuvaa integraatiota koontiversion, hyvien ja huonojen puolien ja tapauskertomuksen kautta. Luvussa kaksi esitetään regressiotestaamisen rooli ja kustannukset jatkuvassa integraatiossa.

2 Jatkuva integraatio

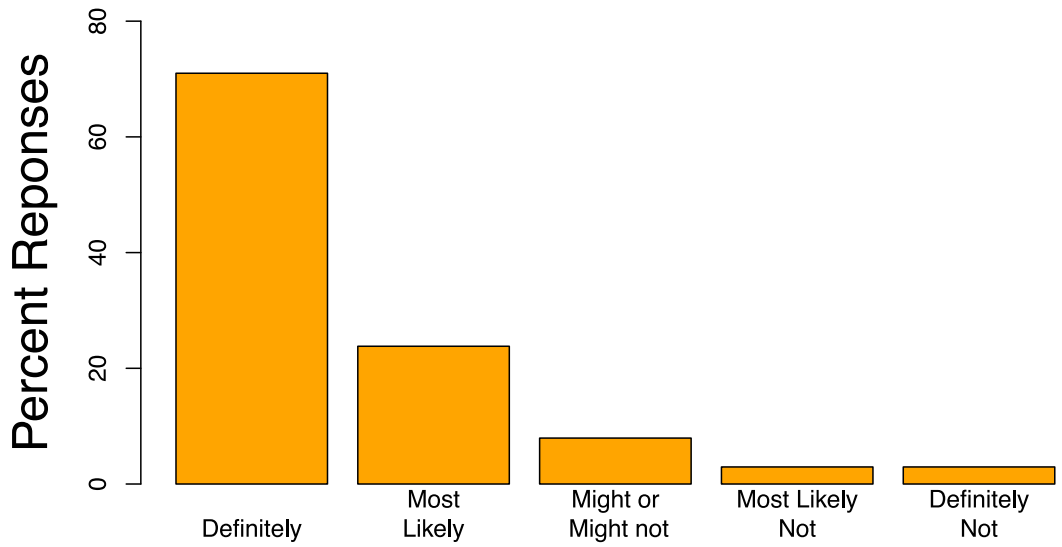
Jatkuva integraatio on suosittu työskentelytapa sekä ohjelmistoteollisuudessa että avoimen lähdekoodin projekteissa (open source projects) [HTH+16]. Ohjelmistokehittäjät pitävät jatkuvasta integraatiosta ja kokevat sen käyttämisen hyödyllisenä. Esimerkiksi erään tutkimuksen mukaan tuhansia ohjelmistokehittäjiä työllistävissä yrityksissä työskentelevät ohjelmistokehittäjät suhtautuvat erittäin myönteisesti jatkuvaan integraatioon. He antoivat asteikolla 1-5 jatkuvalle integraatiolle arvosanaksi 4,13. Suurin osa näistä ohjelmistokehittäjistä myös noudatti jatkuvan integraation suosituksia kommitointitiheydestä, eli he kommitoivat työnsä vähintään kerran päivässä. Tätä tahtia pidetään sopivana kaikille ohjelmistokehittäjille. Toisaalta ohjelmistokehittäjillä on vaikeuksia kommitoida ohjelmistokehityksen päälinjaan ja he käyttävät usein suositusten vastaisesti versionhallinnan sivuhaaroja (branch) [MSB17].



Kuva 2: Avoimen lähdekoodin projektit, joissa käytetään jatkuvaa integraatiota [HTH+16].

Avoimen lähdekoodin projekteissa jatkuvan integraation käyttö on kasvanut hui-

masti. Kuvassa 2 on jatkuvaa integraatiota käyttäviä avoimia projekteja GitHub-pilvipalvelusta vuosilta 2011-2016. Vielä vuonna 2011 projektien määrä on lähellä nollaa, mutta vuosittain jatkuvan integraation käyttö on lisääntynyt lineaarisesti noin sadalla projektilla. Myös avoimen lähdekoodin parissa työskentelevät ohjelmistokehittäjät käyttävät työskentelytapaa mielellään. Eräässä kyselytutkimuksessa kävi ilmi, että noin 70 prosenttia vastaajista käyttäisi ehdottomasti jatkuvaa integraatiota seuraavassa projektissaan. Kiinnostavinta on kuitenkin se, että vain marginaalinen osa vastaajista ei sitä käyttäisi. Tulokset näkyvät kuvassa 3 [HTH+16].



Kuva 3: Käyttäisivätkö avoimen lähdekoodin projekteissa työskentelevät ohjelmistokehittäjät jatkuvaa integraatiota seuraavassa projektissaan [HTH+16].

Trendikkyydestä huolimatta jatkuvan integraation suosion ei odoteta laskevan tulevaisuudessa. Tähän viittaa se, että jatkuvan integraation käyttö madaltaa selvästi kynnystä ottaa osaa avoimen lähdekoodin projektiin. Mitä suositumpi projekti, sitä todennäköisemmin työskentelytapa on käytössä. Projektien parissa työskentelevien on helpompi ottaa puoltopyyntöjä (pull request) vastaan, kun jatkuvan integraation palvelin toimii kommitin oikeellisuuden puolueettomana varmistajana [HTH+16].

Jatkuvaa integraatiota suitsutetaan paljon, mutta katteettomasti. Vaikka jatkuvasta integraatiosta näyttäisi kiistattomasti olevan hyötyä, hyödyn määrittely ja arviointi on vaikeaa [SB13]. Hyvä esimerkki tästä on ohjelmistokehittäjien taipumus vältellä koontiversion epäonnistumista. Huolellisuus ja ohjelmiston päälinjan stabiiliudesta huolehtiminen on tärkeää, kun saman päälinjan parissa työskentelee monia ohjelmistokehittäjiä. Toisaalta kommitointia edeltävään testaamiseen voi kuluttaa liikaa aikaa ja palvelinresursseja, jotka ovat pois muista toiminnoista [HTH+16].

Ohjelmistokehittäjät noudattavat jatkuvan integraation tarkkoja käytänteitä mielellään, jos se on heidän kannaltaan järkevää. Monet heistä eivät näe eroa versionhallinnan päälinjaan ja sivuhaaraan kommitoimisen välillä [MSB17]. Toisaalta kaikki ohjelmistokehittäjät eivät tunne jatkuvaa integraatiota tarpeeksi voidakseen ottaa sitä käyttöön ja harjoittaa suositusten mukaisesti [HTH+16]. Suurin osa jatkuvan integraation ongelmista ja käyttöönoton esteistä on ratkaistavissa käytännön olosuhteita muuttamalla [PRC17].

2.1 Koontiversio

Integraatiossa ohjelmisto kootaan jatkuvan integraation palvelimella, jolloin ajetaan myös ohjelmakoodin sisältämät automatisoidut testit. Palvelin antaa palautetta koonnin ja testaamisen onnistumisesta. Koontiversion epäonnistuessa ohjelmistokehitystiimin kuuluu tehdä välittömästi toimenpiteitä, joilla koontiversion status muuttuu onnistuneeksi [Fow06]. Tosin teoria ja käytäntö eivät aina kohtaa ja havaintojen mukaan koontiversion statuksesta ja laadusta ei viestitä tarpeeksi. Ilmeisesti ohjelmistokehittäjät jättävät kommunikoinnin täysin jatkuvan integraation palvelimen vastuulle [SB13].

Testaaminen ennen kommitointia on tärkeää, jotta koontiversion status pysyy onnistuneena [ERP14]. Näin yhteisen ohjelmiston parissa työskentelevät ohjelmistoke-

hittäjät voivat luottaa työskentelevänsä toimivan ohjelmakoodin varassa [MSB17]. Turvallisuuskriittisissä ohjelmistoissa testaaminen on vieläkin tärkeämpää ja tällaiseen ohjemistoon kohdistuvat muutokset ovat kokonaisvaltaisempia kuin esimerkiksi web-sovelluksissa, joita kehitetään lähempänä tuotantoa ja tuotanto-olosuhteita [LGL+16]. Ketterän testaamisen (agile testing) harjoittaminen korreloi jatkuvan integraation kanssa, mutta kyseessä lienee syy-seuraussuhteen sijaan siitä, että projekteissa omaksutaan kerralla useita ketteriä menetelmiä [SB13].

Koontiversion onnistuminen tarkoittaa kommitin hyväksymistä osaksi ohjelmiston päälinjaa. Koontiversion epäonnistuminen sen sijaan tarkoittaa sitä, että ohjelmiston uusin versio on tavalla tai toisella rikki ja se on korjattava välittömästi. Tähän kuluu luonnollisesti aikaa. Jatkuvan integraation ja jatkuvan integraation palvelimen tarkoitus on löytää virheitä, mutta vain 65 prosenttia epäonnistuneista koontiversioista sisälsi ohjelmointivirheen. Kun virhe löydetään, se voidaan korjata, mutta muut epäonnistumiseen johtaneet syyt sen sijaan hidastavat ohjelmistokehitystä. Virhe voi esimerkiksi sijaita testissä tai testi voi olla epädeterminstinen, eli antaa olosuhteista riippumatta vaihtelevia tuloksia [LIH17].

Koontiversion epäonnistumiseen johtavat syyt voivat olla myös sosiaalisia, riittämätön testaaminen tai kiireinen aikataulu. On myös tutkittu, että ohjelmistokehittäjät luottavat liikaa testien virheettömyyteen [PRC17]. Koontiversioiden epäonnistumiseen johtavien seikkojen analysointi projektin sisällä kannattaa myös rahallisesti [ERP14].

2.2 Jatkuvan integraation hyvät ja huonot puolet

Jatkuvan integraation tärkein hyöty on virheiden löytyminen aikaisin, minkä vuoksi projektien aikataulu on helpommin ennustettavissa [HTH+16]. Sen omaksuminen nopeuttaa ohjelmistotuotantoprosessia ja mahdollistaa ripeämmän julkaisusyklin

[PRC17]. Myös avoimen lähdekoodin projekteissa puoltopyynnöt (pull request) hyväksytään nopeammin. Jatkuva integraatio lisää sekä sisäistä että ulkoista viestintää, mikä mahdollistaa rinnakkaisen ohjelmistokehityksen. Se ei kuitenkaan korvaa ohjelmiston tilasta kommunikointia, vaikka virheiden löytäminen ja versioiden yhteenliittäminen (merge) olisikin helpompaa [SB13].

Testaaminen on olennainen osa jatkuvaa integraatiota, mutta testien kirjoittamiseen ja ylläpitämiseen menee jopa yhtä paljon työaikaa kuin itse ohjelmakoodin kirjoittamiseen [LGL+16] [LIH17]. Lisäkustannuksia voi tulla projektista riippuen ostopalveluista ja välineistä. Testeihin myös luotetaan liikaa, vaikka ne voivat olla epädeterministisiä tai kattavuudeltaan huonoja [PRC17].

Jatkuva integraatio vaatii ohjelmistokehittäjiltä itsekuria, ainakin aluksi. Työyhteisöllä tulee olla hyvä testaamiskulttuuri ja käytänteet on tehtävä selväksi myös uusille työntekijöille [PRC17]. Kuten aiemmin tässä luvussa käytiin läpi, ohjelmistokehittäjät myös välttelevät liikaa koontiversion testien epäonnistumista [HTH+16]. Ohjelmistokehittäjät toteuttaisivatkin integraatiot useammin, mikäli työ olisi pilkottavissa ja järkevästi ajoitettu. Ohjelmiston arkkitehtuurin on siis hyvä koostua itsenäisistä osista ja testaamisessa työkalujen ja prosessien on oltava nopeita ja helppoja. Ottamalla nämä tekijät huomioon on mahdollista luoda ohjelmistokehittäjille paremmat olosuhteet jatkuvan integraation toteuttamiseen. Taulukossa 1 näkyy teemakartoitus ohjelmistokehittäjien esteistä kommitoida päivittäin ohjelmiston versionhallinnan päälinjaan [MSB17].

Jatkuva integraatio on työskentelytapana sellainen, että se rytmittää ohjelmistokehittäjän arkea. Tämän vuoksi jatkuvan integraation prosessien vaivattomuudella on iso vaikutus ohjelmistokehittäjän tyytyväisyyteen ja halukkuuteen toteuttaa jatkuvaa integraatiota [MSB17].

Esimerkiksi automatisoitu testaaminen ja neutraalin jatkuvan integraation palveli-

	Total	**Case study A**	**Case study B**
Activity planning and execution	19	10	9
- Work breakdown	15	7	8
- Teams and responsibilities	6	1	5
- Activity sequencing	13	6	7
System thinking	17	8	9
- Modular and loosely coupled architecture	12	5	7
- Developers must think about the complete system	8	5	3
Speed	19	9	10
- Tools and processes that are fast and simple	15	8	7
- Availability of test environments	9	2	7
- Test selection	7	1	6
- Fast feedback from the integration pipeline	9	5	4
Confidence through test activities	16	9	7
- Test before commit	9	6	3
- Regression tests on the mainline	9	5	4
- Reliability of test environments	6	3	3

Taulukko 1: Teemakartoitus ohjelmistokehittäjien esteistä kommitoida päivittäin ohjelmiston versionhallinnan päälinjaan [MSB17].

men käyttäminen testaamiseen helpottaa virheiden löytymistä, mutta silti ohjelmiston kehitys- ja testaamisvälineet eivät aina ole yhteensopivia ja jatkuvan integraation palvelimen configuroiminen on hankalaa [PRC17]. Jatkuva integraatio voi myös hämärtää eri testitoimintojen rajoja [LGL+16].

2.3 Case: Jatkuva integraatio Googlella

Google on valtava, kansainvälinen tekniikan alan yritys, jossa tuotetaan myös valtavasti ohjelmistoja. Jatkuva integraatio on otettu käyttöön myös Googlella, joten on mielenkiintoista, miten työskentelytapa on tässä mittakaavassa toteutettu. [MGN+17]

Google käyttää testaamiseen testialustaa TAP (Test Automation Platform). TAP suorittaa päivittäin 800 000 koontiversiota, mikä on suuruusluokkaa yksi per sekunti. Se aiheuttaa TAP:lle suorituspaineita, varsinkin kun ajan mittaan sekä testipohjan koko että kommittien saapumistiheys kasvavat ajan myötä lineaarisesti. Ympäri maailmaa sijoittuneet ohjelmistokehitystiimit tasaavat TAP:n kuormitusta, mutta suurin piikki kommiteissa on Yhdysvaltojen Californian osavaltion lounasaikaan. Kuvassa 4 kommittien saapumistiheydet eri vuorokaudenaikoina Californiasta katsoen [MGN+17].

Jatkuva integraatio on Googlella ennen kaikkea hyvin järjestettyä testaamista. [MGN+17]. TAP:n vastuulla on testaaminen sekä ennen että jälkeen kommitoinnin. Kommittien määrä ja koodipohjan koko ovat valtavia, jonka vuoksi koodipohjan analysointiin perustuvien työkalujen käyttö ei onnistu. Testaamisesta ja TAP:sta huolehtii erillinen testaamistiimi, mutta ohjelmistokehittäjät kirjoittavat itse testit ja kokoavat testikokonaisuudet. Testaamisen keskittämistä helpottaa se, että kaikissa testikohteissa käytetään XUnit-tyyppistä testikehystä. Näin kaikkia testikohteita voidaan käsitellä yhtenäisesti [ERP14].

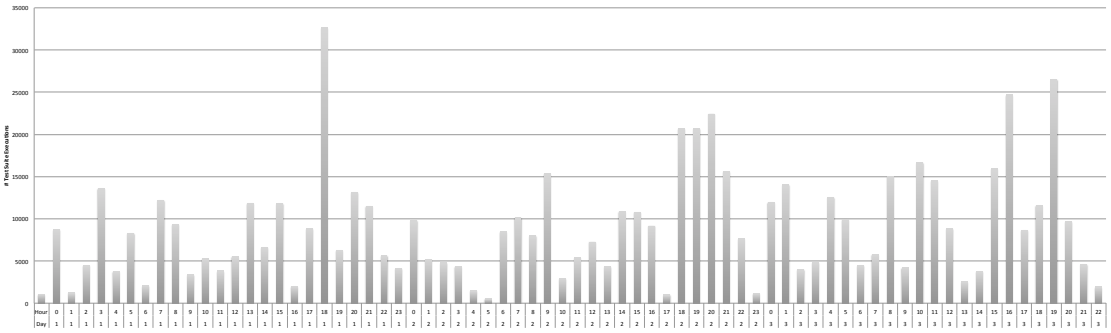


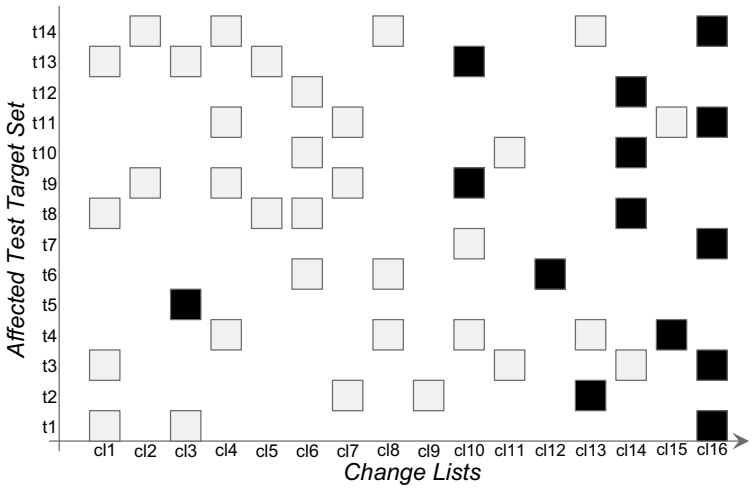
Figure 1: Flow of Incoming Test Suites for Pre-Submit Testing over Three Days.

Kuva 4: Kommitien saapuminen TAP-testialustalle eri vuorokaudenaikoina Yhdysvaltain Californian osavaltion ajassa [ERP14].

Googlen koodipohja on jaettu paketteihin (package), joita vastaavat itsenäisesti koottavat testikohteet (test-target). Ohjelmistokehittäjät liittävät TAP:lle lähtevään kommittiin muutoslistan (changelist), johon on listattu kommitissa muutettuja paketteja vastaavat testikohteet. TAP aloittaa kommitin käsittelyn muutoslistasta. Listan sisällölle etsitään riippuvuusanalyysillä riippuvuudet ja riippuvaisuudet [MGN+17]. Niitä approksimoidaan rankasti [ERP14].

Googlella huomattiin, että TAP ajoi lyhyen ajan sisällä useasti samoja testikohteita. Se kulutti tarpeettomasti resursseja, joten Googlella päädyttiin ratkaisuun, jossa TAP kerää kommitteja 45 minuuttia ennen testikohteiden ajoa. Tätä aikaa kutsutaan nimellä milestone, eli suomeksi virstanpylväs. Kuva 5 havainnollistaa kommitien keräämistä virstanpylvään aikana. Kun aika on kulunut, jokaisesta testikohteesta ajetaan vain viimeisin kommit. Esimerkiksi testikohde t1 esiintyy muutoslistoilla cl1, cl3 ja cl16. Nyt testikohteesta t1 ajetaan muutoslistan cl16 versio.

Tieto testaamisen onnistumisesta lähtee ohjelmistokehittäjälle viesti, joka erittelee tulokset testikohteen tarkkuudella. Hänen tulee korjata epäonnistuneita testikohteita vastaavat paketit välittömästi [ERP14]. Jatkuvan integraation suositusten mukaisesti kommitin lähettämisestä tulisi kulua korkeintaan kymmenen minuuttia pa-



Kuva 5: Kommitien kerääntyminen yhden virstanpylvään aikana. Joka testikohteesta ajetaan vain viimeisimmät kommitit, jotka näkyvät mustina neliöinä [MGN+17].

lautteen saamiseen, mutta Googlella ja muilla suurilla yrityksilläse ei ole realistinen tavoite. TAP:lle lähetetty kommit odottaa parhaassa tapauksessa ajoa kokonaisen virstanpylvään verran. Viipeitä aiheuttavat myös esimerkiksi muistin loppuminen, ohjelmiston kaatuminen ja laitteisto-ongelmat [MGN+17].

3 Regressiotestaaminen jatkuvassa integraatiossa

Jatkuvan integraation myötä samaa ohjelmakoodia muokkaavat kymmenet, sadat tai jopa tuhannet ohjelmistokehittäjät ja heidän kaikkien tulee kommitoida tekemänsä muutokset vähintään kerran päivässä [MSB17]. Jatkuvan integraation skaalautuvuudessa regressiotestaaminen muodostaa pullonkaulan [ERP14]. Ohjelmisto on alati muutoksien kohteena ja ohjelmiston toimivuus ja oikeellisuus vaarantuvat joka muutoksessa. Paras tapa ehkäistä virheiden kasaantumista ja isoja rakenteellisia ongelmia on toteuttaa muutokset ja korjata virheet mahdollisimman tiuhaan [HTH+16].

Regressiotestaamista käytetään koko ohjelmiston toimivuuden ja oikeellisuuden testaamiseen muutoksien jälkeen. Se suoritetaan jatkuvan integraation palvelimella, neutraalissa testiympäristössä. Jatkuvan integraation luonteen vuoksi ohjelmisto regressiotestataan vähintään yhtä monta kertaa kuin ohjelmistolla on kehittäjiä. Ohjelmiston koon ja testien määrän ollessa suuri, regressiotestaamista varten kootaan oma testikokonaisuutensa [ERP14]. Regressiotestaaminen on yleistä, mutta sitä ei käytetä kaikkialla suurten kustannusten vuoksi[LIH17].

Testitoimintojen toimivuus ja sujuvuus on iso osa jatkuvan integraation toimimista ja ohjelmistokehittäjien kokemaa tyytyväisyyttä jatkuvaan integraatioon. [MSB17] Ohjelmistokehitystiimi ei voi työskennellä luotettavasti rikkinäisen ohjelmiston parissa. Siispä jatkuvan integraation palvelimen antama palaute testien läpäisystä toimii vakuutena koko ohjelmistokehitystiimille, että työtä voi jatkaa rauhassa ja se rakentuu toimivan kokonaisuuden päälle. Uudet virheet voidaan paikantaa muutoksen tarkkuudella, eikä kenenkään työtä sotketa turhaan. Testaaminen muuttuu aina vain tärkeämmäksi sitä mukaa, kun lähestytään julkaisua [ERP14]. Jatkuvan integraation palvelimen on tarkoitus tuottaa puolueeton ja luotettava testitulos, joka vertautuu ohjelmiston käyttämiseen oikeissa tuotanto-olosuhteissa. [Fow06]

Regressiotestaamista varten muodostetaan oma testikokonaisuus, sillä testipohjan koko on miltei aina liian suuri suoritettavaksi joka integraatiolla. Yleisimmät tavat muodostaa testikokonaisuus on rajaaminen ja priorisointi. Testikokonaisuuden rajaamisessa kokonaisuus muodostetaan testeistä, jotka tuottavat eniten kulloinkin hyödyllistä tietoa. Koodipohjasta voidaan jokaisen kommitin perusteella tehdä riippuvuusanalyysi, joka paljastaa, mihin muutos vaikuttaa. Kokonaisuuden rajaamista helpottaa ohjelmiston modulaarinen, eli osista muodostuva rakenne, jossa jokainen osa kootaan itsenäisesti. Näin voidaan testaaminen voidaan suorittaa rinnakkain. Testikokonaisuuden priorisoimisella pyritään nopeuttamaan regressiotestaamisesta saatavaa palautetta. Jatkuvan integraation palvelimelta saatava palaute on tärkeä osa jatkuvaa integraatiota [ERP14].

3.1 Kustannukset

Jatkuvan integraation käyttö on kustannustehokkaampaa kuin integraatio erillisenä tuotantovaiheena. Tämä johtuu pitkälti ohjelmistotuotantoprosessin läpinäkyvyyden kasvusta, kun ohjelmistoon syntyneet virheet saatetaan koko ohjelmistokehitystiimin tietoon. Pieniä virheitä on suuria virheitä nopeampi korjata, mikä vähentää kuluvia työtunteja ja ehkäisee aikataulun venymistä. Näin ehkäistään myös rikkonaisen ikkunan syndroomaa, eli ohjelmistokehittäjien taipumusta vältellä suurien virheiden korjaamista [SB13]. Tuotantoon asti ehtineet virheet ovat erityisen kalliita, sillä uudet käyttäjät eivät pidä virheistä sen enempää kuin vanhat käyttäjätkään [LIH17]. Hyvä esimerkki tästä on Pokemon Go -mobiilipeli, joka saavutti maailmanlaajuisen suosion, mutta käyttäjät kyllästyivät nopeasti pelaamista vaikeuttaviin virheisiin.

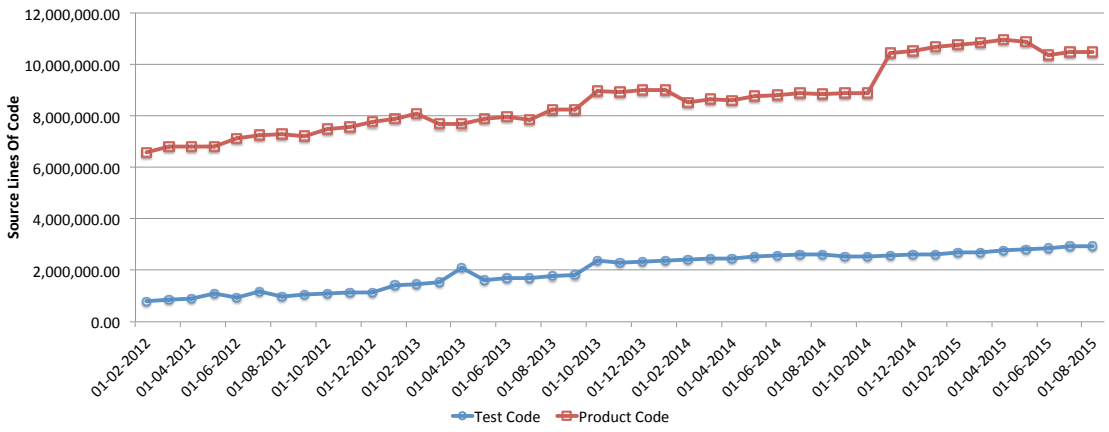
Regressiotestaamisella pyritään estämään virheiden muodostuminen ja nopeuttamaan niiden löytymistä, mutta sen käyttämisen kustannukset eivät saa nousta hyö-

tyjä korkeammiksi. Ohjelmistokehittäjien on koettava, että heidän ponnistelunsa testien kirjoittamisen ja testipohjan ylläpitämisen parissa on sen arvoista [LIH17]. Regressiotestaamisen kustannukset ovat omaa luokkaansa, koska koodipohjat ovat isoja. Niissä on paljon riippuvuuksia ja moduuleita, jotka voivat olla tahoillaan kehitettävinä ja siten aiheuttaa tarpeettomia epäonnistumisia testaamisessa. Ongelmat vähenevät, kun ohjelmistokehittäjät testaavat koodia riittävästi ennen kommitointia. Toisaalta kuten luvussa 2.2 mainittiin, ennen kommitointia tapahtuvasta testaamisesta ei saa muodostua ohjelmistokehitykseen pullonkaulaa. Näin tapahtuu, jos ohjelmistokehittäjä ei uskalla kommitoida koodia ja testaa sitä liikaa. Työajan kulumisen lisäksi liiallinen testaaminen kuluttaa mahdollisen testipalveliman kapasitettia [ERP14].

Jatkuvan integraation skaalautuvuudessa regressiotestaaminen muodostaa pullonkaulan. Koodipohjan, testipohjan ja kommitoinnin määrä kasvavat ja resurssit sitä mukaan hupenevat. Resurssit kuluvat myös liiallisessa kommitointia edeltävässä testaamisessa [ERP14]. Ohjelmiston kokoaminen ja regressiotestaaminen voi johtaa palvelinresurssien riittämättömyyteen. Regressiotestaaminen kuluttaa yhtä paljon työaikaa kuin varsinainen ohjelmiston kehittäminen, sillä regressiotestien tekeminen ja testipohjan ylläpitäminen vie aikaa. Se on paljon, sillä regressiotestaaminen on vain yksi osa ohjelmiston testaamista [MGN+17].

Eräässä tutkimuksessa mitattiin kolmen ja puolen vuoden aikana javaprojektien koodi- ja testipohjan kasvua. Jakson aikana testipohjan koko on kaksinkertaistunut, samalla kun koodipohja on kasvanut vain puolella [LIH17]. Kehitys näkyy kuvassa 6. Suuri koodipohja vaikeuttaa perinteisten testikokoelman käyttämiseen käytettyjen tekniikoiden käyttämistä, sillä ne perustuvat erilaisten analyysityökalujen käyttöön. Koodipohjan kasvu tekee analyysityökaluista raskaita käyttää [ERP14].

Ohjelmiston lisäksi myös testeissä itsessään voi olla virheitä, joiden tunnistaminen ja korjaaminen voi olla hankalaa. [LIH17] Testitoiminnot voi myös rakentaa osin



Kuva 6: Testipohjan osuus koodipohjasta kasvaa ajan myötä. [LIH17]

ulkopuolisen palveluntarjoajan palveluiden varaan, mutta tällöin testejä voi joutua muuttamaan jokaisen palveluntarjoajan tekemän muutoksen myötä. Regressiotestaamisen kustannukset on otettava tarkasti huomioon laskettaessa jatkuvan integraation kustannuksia. [HTH+16]- KATSO PROSENTIT FAILEISTA [LIH17]

Testikokonaisuuden valinnalla on iso vaikutus regressiotestaamisen kustannustehokkuuteen. Projektikohtaiset hyötyjen ja haittojen tutkiminen kannattaa. Samaten koontiversioiden tasolla tehty tarkastelu erilaisten epäonnistumien syistä ja suhteista [LIH17]. Testikokonaisuuden rajaaminen ja testikokonaisuuden priorisointi ovat käytetyimmät tekniikat, mutta sellaisenaan ne sopivat jatkuvan integraation kontekstissa huonosti, sillä niiden käyttö perustuu analysointityökaluihin, joiden soveltaminen suureen koodipohjaan on liian raskasta. Testikokonaisuuksien kustannustehokkuutta voi kasvattaa strategialla, jossa yhdistellään näitä tekniikoita. Isoin parannus saadaan aikaan sillä, että ennen kommitointia suoritettavassa testaamisessa rajataan testikokonaisuuksia aikaikkuna-historiadata-menetelmällä. Kommitoinnin jälkeen sen sijaan priorisointi parantaa merkittävästi hyödyllisen palautteen saamisaikoihin [ERP14].

4 Yhteenveto

Jatkuva integraatio on hyväksi todettu tapa tuottaa laadukaita ohjelmistoja. Se on suosittu työskentelytapa, jota käytetään laajalti sekä ohjelmistoyrityksissä että avoimen lähdekoodin projekteissa. Ohjelmistokehittäjät pitävät jatkuvasta integraatiosta ja kokevat sen hyödyllisenä. Toisaalta käytännön toteutuksessa on vielä parantamisen varaa, sillä ohjelmistokehittäjät eivät aina koe jatkuvan integraation toteutustapoja käytännöllisinä. Tämän vuoksi ohjelmistoyrityksissä harvemmin noudatetaan jatkuvan integraation täsmällistä määritelmää. Myös tietoa jatkuvasta integraatiosta voisi olla enemmän ja ohjelmistokehittäjät motivoituneempia.

Jatkuvan integraation palvelimella kootaan ohjelmiston uusin versio ja kokoamisen tulosta kutsutaan koontiversioksi. Koonnin yhteydessä ohjelmisto regressiotestataan. Koontiversio onnistuu, kun kokoaminen ja testaaminen onnistuu. Tulos lähtetetään koko ohjelmistokehitystiimille. Mikäli koontiversio epäonnistuu, sen kommitoineen ohjelmistokehittäjän odotetaan korjaavan virheet välittömästi.

Tieto jatkuvan integraation hyödyistä ja haitoista lisääntyy varmasti lähivuosina, kun sitä tutkitaan lisää. Hyödyistä kiistattomin on ohjelmointivirheiden löytyminen ajoissa. Se vähentää koodin uudelleenkirjoittamistyötä sekä aikaa, joka menisi ajan myötä kasvaneen virheellisen koodin korjaamiseen. Näin ohjelmistotuotantoprojektit pysyvät paremmin aikataulussa ja voivat julkaista uusia versioita nopeammin kuin projektit, joissa jatkuva integraatio ei ole käytössä. Koontiversion onnistumisen tiedottaminen lisää tietoa ohjelmiston tilasta ja ainakin ohjelmistokehittäjät itse kokevat, että jatkuva integraatio parantaa ohjelmiston laatua.

Jatkuvan integraation palvelimella suoritettava regressiotestaaminen validoi muutokset. Se on tärkeää, jotta suurikin ohjelmistokehitystiimi voi työskennellä yhdessä saman koodipohjan parissa. Regressiotestit valikoidaan kustannussyistä niin, että ohjelmistosta testataan lähinnä osat, joihin kommitin tuoma muutos vaikuttaa.

Regressiotestaaminen vie varsinkin suurissa ohjelmistotuotantoprojekteissa niin paljon palvelinresursseja, että testikokoelman tiukka karsiminen on tarpeen. Palvelinresurssien lisäksi kuluu runsaasti myös henkilöresursseja, eli ohjelmistokehittäjän työaikaa. Testikokonaisuuden kehittäminen voi viedä jopa puolet ohjelmistokehittäjän ajasta, vaikka regressiotestaaminen on vain yksi osa ohjelmiston testaamista.

Jatkuva integraatio vähentää ohjelmistokehityksen riskejä, kun ohjelmisto altistetaan riskialttiille muutoksille monta kertaa päivässä. Riskit realisoituvat ja niihin reagoidaan nopeasti, jolloin lopputuloksena on ajantasainen ja läpinäkyvä ohjelmisto. Regressiotestaamisen rooli jatkuvassa integraatiossa on tärkeä, sillä se validoi integraatiot ja sen seurauksena virheet on helppo jäljittää kommitin tarkkuudella.

Lähteet

ERP14 Elbaum, S., Rothermel, G. ja Penix, J., Techniques for improving regression testing in continuous integration development environments. *The 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. ACM, November 16-21 2014, sivut 235–245.

Fow06 Fowler, M., Continuous integration, saatavissa: https://martinfowler.com/articles/continuousIntegration.html, May 01 2006. [viitattu 31.10.2017].

HTH+16 Hilton, M., Tunnell, T., Huang, K., Marinov, D. ja Dig, D., Usage, costs, and benefits of continuous integration in opensource projects. *ASE'16*, September 3-7 2016, sivut 426–437.

LGL+16 Li, N., Guo, J., Lei, J., Li, Y., Rao, C. ja Cao, Y., Towards agile testing for railway safetycritical software. *XP16 Workshops Proceedings of the Scientific Workshop Proceedings of XP2016*. ACM, May 24-27 2016.

LIH17 Labuschagne, A., Inozemtseva, L. ja Holmes, R., Measuring the cost of regression testing in practice: A study of java projects using continuous integration. *ESEC/FSE'17*. ACM, September 4-8 2017, sivut 821–830.

MGN+17 Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R. ja Micco, J., Taming google-scale continuous testing. *39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE/ACM, May 20-28 2017, sivut 233–242.

MSB17 Mårtensson, T., Ståhl, D. ja Bosch, J., Continuous integration impediments in large-scale industry projects. *2017 IEEE International Conference on Software Architecture*. IEEE, April 3-7 2017, sivut 169–178.

PRC17 Pinto, G., Rebouças, M. ja Castor, F., Inadequate testing, time pressure, and (over) confidence: A tale of continuous integration users. *10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE/ACM, 2017, sivut 74–77.

SB13 Ståhl, D. ja Bosch, J., Experienced benefits of continuous integration in industry software product development: A case study. *The 12th IASTED International Conference on Software Engineering*, March 4 2013, sivut 736–743.