hyväksymispäivä arvosana

arvostelija

Jatkuva integraatio

Piia Hartikka

Helsinki 17.12.2017

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	17.12.2017	20 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 huomattaisiin ja korjattaisiin ajoissa. Näin ohjelmistotuotannon projektit muuttuvat aikataulultaan ja kustannuksiltaan ennustettavammiksi. Integraatioon kuuluu ohjelmiston kokoaminen erillisellä jatkuvan integraation palvelimella. Kokoamisen yhteydessä ohjelmisto myös testataan. Yleisin tapa tässä tilanteessa on regressiotestaaminen, joka keskittyy testaamaan muutoksen vaikutusta ohjelmistoon ja sen oikeellisuuteen. Tässä tutkielmassa käsitellään jatkuvaa integraatiota ja regressiotestaamista. Jatkuvasta integraatiosta käydään läpi koontiversio (build) ja jatkuvan integraation hyvät ja huonot puolet. Esimerkkitapauksena on hakukonejätti Google. Lisäksi tutkielmassa tarkastellaan regressiotestaamisen merkitystä jatkuvalle integraatiolle ja sen kustannuksille.

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

Avainsanat — Nyckelord — Keywords
jatkuva integraatio, regressiotestaaminen

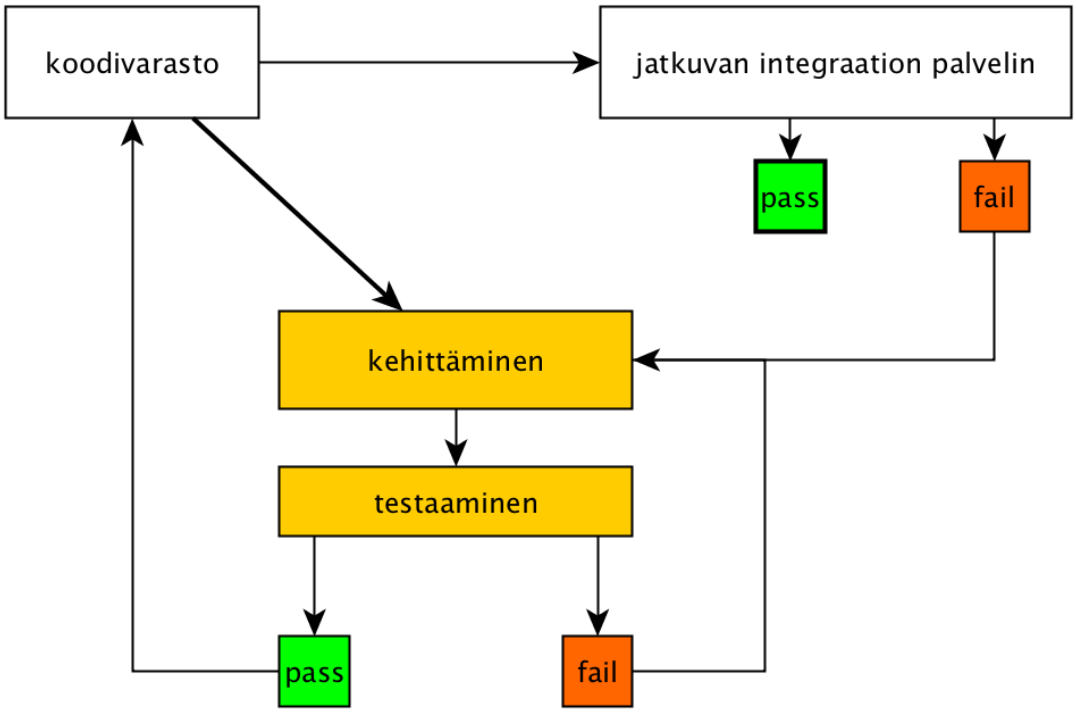
Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto

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. Ohjelmistokehitystä havainnollistaa kuva 1. Jatkuvassa integraatiossa jokainen kehityssykli loppuu vasta sitten, kun jatkuvan integraation palvelin ilmoittaa kommitin koontiversion (build) onnistumisesta koko ohjelmistokehitystiimille. [Fow06]



Kuva 1: Jatkuva integraatio käytännössä.

Jatkuvan integraation testitoiminnot voidaan jakaa karkeasti kahteen luokkaan: tes-

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

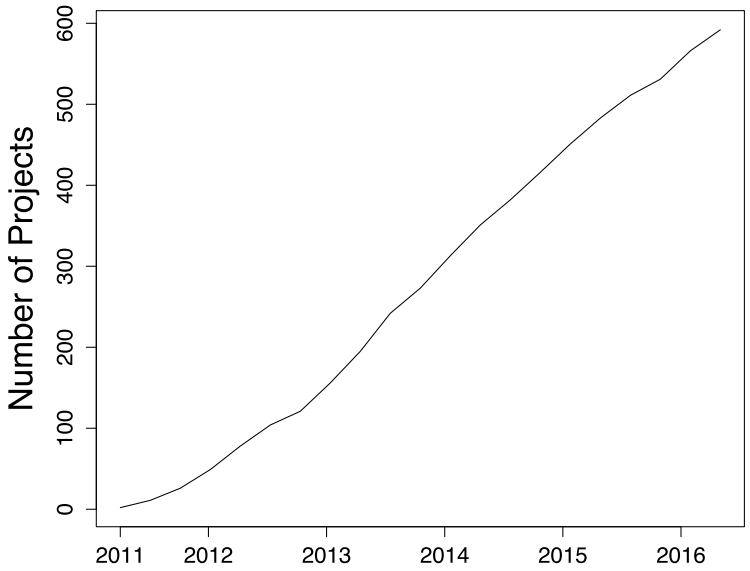
Regressiotestaamisella varmistetaan ohjelmiston oikeellisuus muutoksen jälkeen. Regressiotestaaminen voidaan toteuttaa esimerkiksi valitsemalla kokemuksen perusteella sopiva joukko yksikkö-, integraatio- ja järjestelmätestejä. Kaikkien mahdollisten testien suorittaminen ei ole kannattavaa, sillä testaaminen kuluttaa palvelinresursseja ja viivyttää ohjelmiston tilasta kertovan palautteen saapumista. Ohjelmiston kokoamisen ja testien ajamisen pitäisi suosituksen mukaan kestää alle kymmenen minuuttia. Testiajojen määrää voi rajoittaa 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 [SB13]. 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. 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

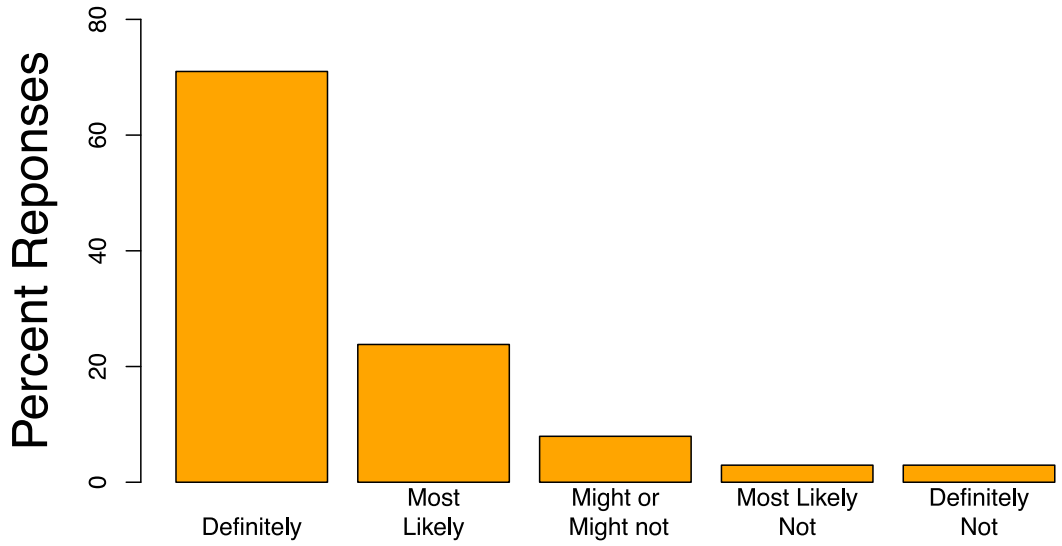
Jatkuva integraatio on suosittu työskentelytapa sekä ohjelmistoteollisuudessa että avoimen lähdekoodin projekteissa (open source projects) [HTH+16]. Ohjelmistokehittäjät pitävät jatkuvasta integraatiosta ja kokevat sen käytön hyödyllisenä [MSB17]. Esimerkiksi erään tutkimuksen mukaan tuhansia ohjelmistokehittäjiä 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. Tahtia pidettiin sopivana kaikille ohjelmistokehittäjille. Toisaalta ohjelmistokehittäjillä on vaikeuksia kommitoida ohjelmistokehityksen päälinjaan ja he käyttävät usein suositusten vastaisesti versionhallinnan sivuhaaroja (branch).



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

Avoimen lähdekoodin projekteissa jatkuvan integraation käyttö on kasvanut huimasti. 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 vuodessa. Myös avoimen lähdekoodin parissa työskentelevät ohjelmistokehittäjät käyttävät jatkuvaa integraatiota mielellään. Kyselytutkimuksessa (katso kuva 3) 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. [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 jatkuva integraatio 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 osin katteettomasti. Vaikka jatkuvasta integraatiosta näyttäisi 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 tarpeenmukaista [MSB17]. Monet heistä eivät näe eroa versionhallinnan päälinjaan ja sivuhaaraan kommitoimisen välillä. Toisaalta kaikki ohjelmistokehittäjät eivät tunne jatkuvaa integraatiota tarpeeksi voidakseen ottaa sitä käyttöön ja harjoittaa sitä 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 [Fow06]. Palvelin antaa palautetta koonnin ja testaamisen koko ohjelmistokehitystiimille. Koontiversion epäonnistuessa ohjelmistokehittäjän kuuluu tehdä välittömästi toimenpiteitä, joilla koontiversion status muuttuu onnistuneeksi. Tässä teoria ja käytäntö eivät aina kohtaa ja havaintojen mukaan koontiversion statuksesta ja ohjelmiston laadusta ei viestitä tarpeeksi [SB13]. Ilmeisesti ohjelmistokehittäjät jättävät kommunikoinnin täysin jatkuvan integraation palvelimen vastuulle.

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. Mikäli virhe löydetään, se voidaan korjata, kun taas muut epäonnistumiseen johtaneet syyt hidastavat ohjelmiston kehittämistä. Virhe voi esimerkiksi sijaita testissä tai testi voi olla epädeterministinen, eli se antaa epäluotettavia tuloksia. [LIH17]

Koontiversion epäonnistumiseen johtavat syyt voivat olla myös sosiaalisia, kuten esimerkiksi riittämätön testaaminen tai kiireinen aikataulu [PRC17]. On myös tutkittu, että ohjelmistokehittäjät luottavat liikaa testien virheettömyyteen. Koontiversioiden epäonnistumiseen johtavien seikkojen projektikohtainen analysointi kannattaa rahallisesti, sillä turhaan epäonnistuneet koontiversiot kuluttavat resursseja [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 [SB13]. Se ei kuitenkaan korvaa ohjelmiston tilasta kommunikointia, vaikka virheiden löytäminen ja versioiden yhteenliittäminen (merge) olisikin helpompaa.

Jatkuva integraatio vaatii ohjelmistokehittäjiltä ainakin aluksi itsekuria [PRC17]. Sitä tukee myös hyvä testaamiskulttuuri ja käytänteet, joiden pariin uudet työntekijät muistetaan tutustuttaa. Kuten aiemmin tässä luvussa käytiin läpi, ohjelmistokehittäjät välttelevät liikaa koontiversion testien epäonnistumista ja tähän ilmiöön voidaan parhaiten puuttua työyhteisön tasolla [HTH+16].

Testaaminen on olennainen osa jatkuvaa integraatiota [LGL+16]. Testien kirjoittamiseen ja ylläpitämiseen meneekin paljon enemmän työaikaa kuin itse ohjelmakoodin kirjoittamiseen [LIH17]. Lisäkustannuksia voi tulla projektista riippuen ostopalveluista ja välineistä. Testeihin luotetaan liikaa, vaikka ne voivat olla epädeterminististisiä tai kattavuudeltaan huonoja. Jatkuva integraatio voi hämärtää eri testitoimintojen ja niiden käyttötarkoitusten rajoja ohjelmistokehittäjän mielessä.

Ohjelmistokehittäjät toteuttaisivat integraatiot useammin, mikäli työ olisi pilkottavissa ja järkevästi ajoitettu (katso taulukko 1). Ohjelmiston arkkitehtuurin on siis jatkuvan integraation kannalta hyvä koostua itsenäisistä osista. Lisäksi testaamisen työkalujen ja prosessien on oltava nopeita ja helppoja, jotta niiden käyttö olisi mahdollisimman luontevaa. Nämä ovat periaatteessa helppoja tapoja luoda sopivat olosuhteet jatkuvan integraation käytölle, mutta käytännössä kaiken muuttaminen kerralla olisi haasteellista. [MSB17]

Jatkuva integraatio on työskentelytapana sellainen, että se rytmittää ohjelmistokehittäjän arkea [MSB17]. Tämän vuoksi jatkuvan integraation prosessien vaivat-

	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].

tomuudella on iso vaikutus ohjelmistokehittäjän tyytyväisyyteen ja halukkuuteen toteuttaa jatkuvaa integraatiota. Automatisoitu testaaminen ja neutraalin jatkuvan integraation palvelimen käyttäminen testaamiseen helpottaa virheiden löytymistä, mutta silti ohjelmiston kehitys- ja testaamisvälineet eivät aina ole yhteensopivia [LGL+16]. Avoimen lähdekoodin projekteissa korostuu jatkuvan integraation palvelimen konfigurointi, joka voi osoittautua erittäin haasteelliseksi tehtäväksi [PRC17].

2.3 Case: Jatkuva integraatio Googlella

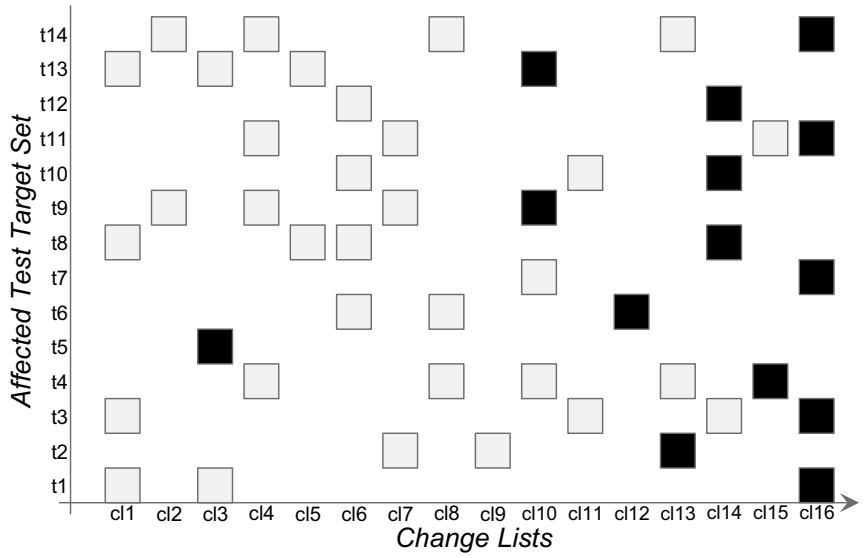
Google on valtava, kansainvälinen tekniikan alan yritys, jossa tuotetaan myös valtavasti ohjelmistoja. Jatkuva integraatio on otettu siellä viime vuosina vaiheittain käyttöön. Tässä mittakaavassa jatkuvan integraation järjestäminen on mielenkiintoinen esimerkki. [MGN+17]

Google käyttää testaamiseen TAP-nimistä (Test Automation Platform) testialustaa. TAP suorittaa päivittäin noin 800 000 koontiversiota, mikä on suuruusluokkaa yksi sekunnissa. Se aiheuttaa TAP:lle suorituspaineita, varsinkin kun ajan mittaan sekä testipohjan koko että kommittien saapumistiheys ovat kasvaneet lineaarisesti. Ympäri maailmaa sijoittuneet ohjelmistokehitystiimit tasaavat TAP:n kuormitusta, mutta suurin piikki kommiteissa on lounasaikaan USA:n Californian osavaltiossa. Seuraavaksi suurin piikki on työpäivän loppuessa. [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 [ERP14]. 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.

Googlen koodipohja on jaettu paketteihin (package), joita vastaavat itsenäisesti koottavat testikohteet (test-target) [MGN+17]. 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. Niitä tosin approksimoidaan voimakkaasti [ERP14].



Kuva 4: Kommitien kerääntyminen yhden virstanpylvään aikana. Testikohteista ajetaan vain viimeisimmät kommitit, jotka on kuvassa väritetty mustaksi. [MGN+17]

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 4 havainnollistaa kommitien keräämistä virstanpylvään aikana. Kun aika on kulunut, jokaisesta testikohteesta ajetaan vain viimeisin kommit. Esimerkiksi testikohde t1 esiintyy muutoslistoilla cl1, cl3 ja cl16.

Testikohteesta t1 ajetaan lopuksi vain muutoslistan cl16 versio. [MGN+17]

Testaamisen onnistumisesta lähtee ohjelmistokehittäjälle viesti, joka erittelee tulokset testikohteen tarkkuudella [ERP14]. Ohjelmistokehittäjän on korjattava epäonnistuneita testikohteita vastaavat paketit välittömästi. Jatkuvan integraation suositusten mukaisesti kommitin lähettämisestä tulisi kulua korkeintaan kymmenen minuuttia palautteen saamiseen. Googlella, kuten muillakaan suurilla yrityksillä, se 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]. Jokainen kommit johtaa ohjelmiston kokoamiseen ja testaamiseen jatkuvan integraation palvelimella, jolloin regressiotestaaminen muodostaa helposti jatkuvaan integraatioon pullonkaulan [ERP14]. Regressiotestaamista käytetään jatkuvan integraation palvelimella koko ohjelmiston toimivuuden ja oikeellisuuden testaamiseen muutoksien jälkeen. Ohjelmiston ollessa alati muutoksien kohteena sen toimivuus ja oikeellisuus vaarantuvat joka muutoksessa, minkä vuoksi regressiotestaaminen on tärkeä osa jatkuvaa integraatiota. Paras tapa ehkäistä virheiden kasaantumista ja isoja rakenteellisia ongelmia on toteuttaa muutokset ja korjata virheet mahdollisimman tiuhaan [HTH+16]. Regressiotestaaminen on yleistä, mutta sitä ei käytetä kaikkialla suurten kustannusten vuoksi[LIH17].

Ohjelmistokehittäjien kokema tyytyväisyys jatkuvaa integraatiota kohtaan on osaltaan riippuvainen testitoimintojen toimivuudesta ja sujuvuudesta [MSB17]. Esimerkiksi ohjelmistokehitystiimi ei voi työskennellä luotettavasti rikkinäisen ohjelmiston parissa. Jatkuvan integraation palvelimen antama palaute onnistuneesta testaamisesta toimii vakuutena koko ohjelmistokehitystiimille, että työtä voi jatkaa rauhassa ja se rakentuu toimivan kokonaisuuden päälle. Mikäli testejä epäonnistuu, ohjelmointivirheet on mahdollista jäljittää kommitin ja testin tarkkuudella. Virheet korjataan, eikä periaatteessa kenenkään työtä sotketa. Ohjelmiston julkaisun lähestyessä regressiotestaaminen muuttuu aina vain tärkeämmäksi, koska jatkuvan integraation palvelimen on tarkoitus olla neutraali testiympäristö ja tuotannossa ilmenevät virheet karkoittavat asiakkaita [ERP14].

Regressiotestaamista varten muodostetaan oma testikokonaisuus, sillä testipohjan

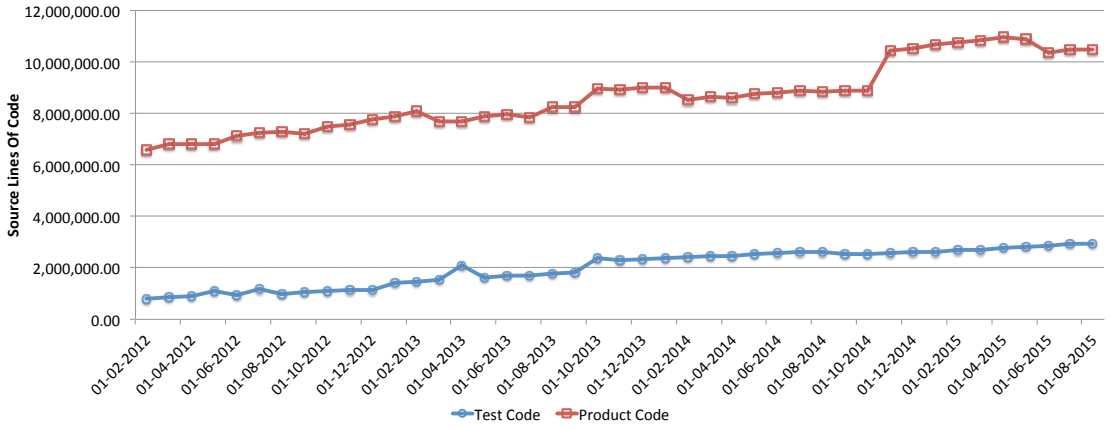
koko on miltei aina liian suuri suoritettavaksi joka integraatiolla. Yleisimmät tavat muodostaa testikokonaisuus on rajaaminen ja priorisointi. Testikokonaisuuden rajaamisessa kokonaisuus muodostetaan testeistä, jotka tuottavat eniten kulloinkin hyödyllistä tietoa. Koodipohjasta voidaan esimerkiksi tehdä jokaisella kommitilla riippuvuusanalyysi, joka paljastaa, mihin uusi muutos vaikuttaa. Kokonaisuuden rajaamista helpottaa ohjelmiston modulaarinen, eli osista muodostuva rakenne, jossa osat kootaan itsenäisesti. Näin testaamista voidaan myös suorittaa rinnakkain. Testikokonaisuuden priorisoimisella pyritään nopeuttamaan regressiotestaamisen palautteen saamiseen kuluvaa aikaa. 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 häiritseviin virheisiin.

Regressiotestaamisella pyritään nopeuttamaan virheiden löytymistä ja korjaamista, mutta sen käyttämisen kustannukset eivät saa nousta hyötyjä suuremmiksi. Ohjelmistokehittäjien on koettava, että heidän ponnistelunsa testien kirjoittamisen ja testipohjan ylläpitämisen parissa on tarpeellista [LIH17]. Regressiotestaamisen kus-

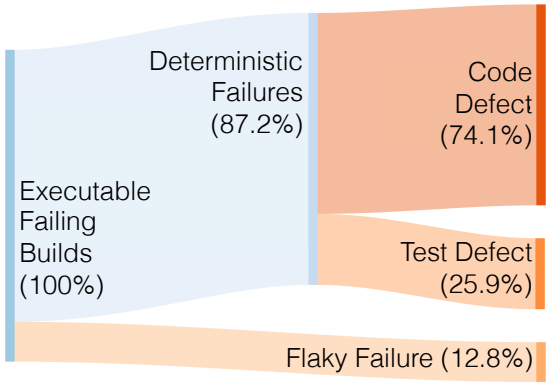
tannuksiin vaikuttavat erityisesti valtavat koodi- ja testipohjat. Niissä on paljon riippuvuuksia ja moduuleita, jotka voivat olla tahoillaan kehitettävinä. Tästä aiheutuu tarpeettomia epäonnistumisia testaamisessa. Ongelmat vähenevät, kun ohjelmistokehittäjät testaavat koodia riittävästi ennen kommitointia. Toisaalta kuten luvussa 2.2 mainittiin, ennen kommitointia tapahtuvasta testaamisesta ei saa muodostua ohjelmistokehitykseen toista pullonkaulaa. Ohjelmistokehittäjä eivät välttämättä uskalla kommitoida koodia ilman liiallista testaamista, josta aiheutuu sekä ylimääräisiä työtunteja että palvelinresurssien kulumista [ERP14].



Kuva 5: Testipohjan osuus koodipohjasta kasvaa ajan myötä [LIH17].

Luvun alussa todettiin, että suurilla määrillä kommitteja regressiotestaaminen muodostaa helposti pullonkaulan jatkuvaan integraatioon. Koodipohjan, testipohjan ja kommitoinnin kasvun mukana hupenevat myös käytössä olevat resurssit [ERP14]. 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 (katso kuva 5) mitattiin kolmen ja puolen vuoden aikana javaprojektien koodi- ja testipohjan kasvua [LIH17]. Jakson aikana testipohjan koko on kaksinkertaistunut, samalla kun koodipohja on kasvanut vain puolella. Suuri koodipohja vaikeuttaa testikokoelman muodostamista, sillä perintei-

set menetelmät siihen perustuvat koodipohjan analysoimiseen ja koodipohjan kasvu tekee siitä liian raskasta.



Kuva 6: Koontiversion epäonnistumiseen johtaneet syyt [LIH17].

Myös testeissä itsessään voi olla virheitä, joiden tunnistaminen ja korjaaminen on vaikeaa [LIH17]. Näin on erityisesti silloin, jos testitoiminnot on rakennettu osin ulkopuolisen palveluntarjoajan palveluiden varaan ja testejä voi joutua muuttamaan jokaisen palveluntarjoajan muutoksien myötä [HTH+16]. Regressiotestaamisen kustannukset on otettava tarkasti huomioon laskettaessa jatkuvan integraation kustannuksia. Jatkuvan integraation palvelimella koontiversioista 82% onnistuu, jolloin vastaavasti 12% epäonnistuu. Epäonnistuneista koontiversioista (katso kuva 6) 35% on turhia ja resursseja hukkaavia tapahtumia. Koontiversio joko kaatuu, testit ovat epädeterministisiä tai niissä on virhe. Näin ollen kaikista jatkuvan integraation palvelimen koonneista noin 4% olisi voitu välttää kokonaan. Lisäksi nämä turhaksi luo-

kitellut epäonnistumiset johtavat useisiin peräkkäisiin epäonnistuneisiin koonteihin, kun virhettä yritetään korjata.

Testikokonaisuuden valinta suuntaa regressiotestaamisen kustannustehokkuutta. Sopivaa testikokonaisuutta ja sen muodostamismenetelmää kannattaa tutkia projektikohtaisesti. Myös jatkuvan integraation palvelimen tilastot ovat tärkeä osa sopivan tavan löytämisessä [LIH17]. Testikokonaisuuden rajaaminen ja testikokonaisuuden priorisointi ovat käytetyimmät muodostamismenetelmät. Ne eivät kuitenkaan sovi sellaisenaan jatkuvan integraation kehyksessä, sillä niiden käyttö perustuu pitkälti koodin analysointityökaluihin, joiden soveltaminen suureen koodi- ja testipohjaan on raskasta. Testikokonaisuuksien kustannustehokkuutta voi sen sijaan kasvattaa strategialla, joka yhdistelee eri menetelmiä. Esimerkiksi Googlella isoin parannus saatiin aikaan, kun ennen kommitointia suoritettavassa testaamisessa rajataan testikokonaisuuksia tietoa aiemmista testituloksista. Kommitoinnin jälkeen sen sijaan priorisointi parantaa merkittävästi hyödyllisen palautteen saamisaikaa [ERP14].

4 Yhteenveto

Jatkuva integraatio on hyväksi todettu tapa tuottaa ohjelmistoja. Se on suosittu työskentelytapa, jota käytetään laajalti sekä ohjelmistoyrityksissä että avoimen lähdekoodin projekteissa. Ohjelmistokehittäjät pitävät jatkuvasta integraatiosta ja kokevat sen hyödyllisenä, mutta toisaalta käytännön toteutuksessa on vielä parantamisen varaa. Ohjelmistokehittäjät eivät aina koe jatkuvan integraation toteutustapoja käytännöllisinä, minkä vuoksi ohjelmistoyrityksissä harvemmin noudatetaan jatkuvan integraation täsmällistä määritelmää. Jatkuvan integraation harjoittaminen on joskus kiinni myös tietotaidosta ja motivaatiosta.

Jatkuvan integraation palvelimella kootaan ja testataan ohjelmiston uusin versio ja kokoamisen tulosta kutsutaan koontiversioksi. Ohjelmistokehittäjät saavat jatkuvan integraation palvelimelta palautetta koonnin onnistumisesta ja tieto saatetaan myös kaikkien asianomaisten tietoon. On tärkeää, että ohjelmiston tila tunnetaan reaaliaikaisesti, jotta ohjelmistokehittäjät voivat työskennellä yhdessä. Koontiversio onnistuu, kun kokoaminen ja testaaminen onnistuu. Epäonnistuneen koontiversion kommitoineen ohjelmistokehittäjän odotetaan korjaavan virheet välittömästi.

Jatkuvan integraation hyödyt ja haitat ovat vielä suhteellisen tuntemattomia, sillä niitä on tutkittu kyselytutkimuksilla, joiden tulokset ovat olleet ristiriitaisia. Selkein ja kiistattomin hyöty 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 onnistumisesta tiedottaminen lisää tietoa ohjelmiston tilasta ja ainakin ohjelmistokehittäjät itse kokevat, että jatkuva integraatio parantaa ohjelmiston laatua.

Jatkuvan integraation palvelimella suoritettava regressiotestaaminen validoi kom-

mitoidut muutokset. Näin 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 helposti 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 kehitysvaiheensa tasolla oleva ohjelmisto. Regressiotestaamisen rooli jatkuvassa integraatiossa on tärkeä, sillä se validoi muutokset 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.