Tampereen yliopisto

Ville Karvonen

PYTHON-KEHITYSYMPÄRISTÖN AUTOMATISOINTI

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Kesäkuu 2021

TIIVISTELMÄ

VILLE KARVONEN: Python-kehitysympäristön automatisointi
Tampereen yliopisto
Diplomityö, 41 sivua
Kesäkuu 2021
Diplomi-insinöörin tutkinto, Tietotekniikka, DI
Pääaine: Ohjelmistotuotanto
Tarkastajat: Yliopistolehtori Terhi Kilamo ja Yliopistolehtori Timo Poranen

Avainsanat: Python, kehitysympäristö, jatkuva toimitus, gitlab, gitlab ci/cd

Ohjelmistokehitys on muuttunut kahden viime vuosikymmenen aikana paljon. Aikaisemmin kehitys oli hidasta ja ohjelmistoprojektit pitkiä. Nykyään kehitys on nopeampaa ja projektit lyhytikäisempiä. Tähän on vaikuttanut ohjelmistosuunnittelun kehittyminen ja tietotekniikan halpeneminen, joka on mahdollistanut tehokkaampien kehitysprosessien käyttöönoton.

Tässä työssä tavoitteena oli kehittää perinteiseen ohjelmistokehitykseen perustuvaa Python-kehitysympäristöä vastamaan paremmin nykyaikaista ohjelmistokehitystä. Kehityksessä sovellettiin jatkuvan toimituksen käytäntöä, joka ratkaisee perinteisen ohjelmistokehityksen ongelmia automatisoinnilla. Työssä tutustuttiin käytäntöön ja sen tarvitsemiin työkaluihin perusteellisemmin. Lopuksi pohdittiin miten hyvin kehitysympäristö soveltaa jatkuvan käytäntöä ja miten sitä voisi hyödyntää paremmin.

Työn lopputuloksena kehitysympäristöön lisättiin monitorointi sekä testaus ja käyttöönotto vaiheet automatisointiin julkaisuputkilla. Tulokset olivat positiivisia, vaikka kehitysympäristö ei vastannut täysin jatkuvan toimituksen määrityksiä. Toteutettua kehitysympäristöä voidaan hyödyntää myös muihin Pythonilla tehtyihin ohjelmistoprojekteihin.

ABSTRACT

VILLE KARVONEN: Automation of Python development enviroment
Tampereen University
Master of Science (Technology), 41 pages
June 2021
Information Technology, MSc
Major: Software Engineering
Examiners: University Lecturer Terhi Kilamo and University Lecturer Timo Poranen

Keywords: Python, development environment, continuous delivery, gitlab, gitlab ci/cd

Software development has changed a lot in last two decades. In the past, development was slow and software projects were long. Today, development is faster, and projects are shorter. This has been made possible due to the development of software designing and the cheapening of computation resources, which has helped to introduce more efficient development processes.

The goal of this work was to develop a Python development environment based on traditional software development to better match modern software development. The development applied practice of continuous delivery, which solves the problems of traditional software development through automation. The work explored practices and tools needed in continuous delivery in more depth. Finally, it was analyzed how well the development environment applies continuous delivery practices and how it could be better utilized.

As an end result, monitoring was added to the development environment and testing and deployment phases were automated with pipelines. The results were positive, although the development environment did not fully meet the specifications of continuous delivery. The implemented development environment can be utilized for other software projects made with Python.

ALKUSANAT

Isot kiitokset Terhi Kilamolle hyvästä ohjauksesta ja Timo Poraselle työn tarkastamisesta.

Kiitokset myös Hatelle sekä Jonille tuesta ja hyvistä tsempeistä työn aikana.

Tampereella, 21.6.2021

Ville Karvonen

SISÄLLYSLUETTELO

1. Johdanto . . . 1
2. Jatkuva toimitus . . . 3
2.1 Periaatteet . . . 3
2.2 Perusta . . . 4
2.2.1 Jatkuva testaus . . . 5
2.2.2 Jatkuva integrointi . . . 7
2.2.3 Julkaisuputki . . . 9
2.3 Hyödyt . . . 10
2.4 Haasteet . . . 11
2.5 Erot jatkuvaan käyttöönottoon . . . 12
3. Jatkuvan toimituksen työkalut . . . 13
3.1 Jatkuva toimitus ja -käyttöönotto . . . 14
3.1.1 GitLab CI/CD . . . 14
3.1.2 Jenkins . . . 16
3.2 Monitorointi . . . 16
3.2.1 Zabbix . . . 16
3.2.2 Sentry . . . 18
4. Työn kohde . . . 20
4.1 Esittely . . . 20
4.2 Kehitysympäristö . . . 21
4.2.1 Kehitysprosessi . . . 21
4.2.2 Asennus- ja päivitysprosessi . . . 22
4.3 Ongelmat . . . 23
5. Python-kehitysympäristön automatisointi . . . 24
5.1 Työkalujen valintakriteerit . . . 24
5.2 Monitoroinnin kehittäminen . . . 25
5.2.1 Järjestelmätason monitorointi . . . 25
5.2.2 Ohjelmatason monitorointi . . . 27
5.3 Testauksen automatisointi . . . 28
5.3.1 GitLab CI/CD asentaminen . . . 28

5.3.2 Automaatiotestit 29
5.4 Käyttöönoton automatisointi 30
5.5 Asennus- ja päivitysprosessin parantaminen 30
5.5.1 Anittan ja sen moduulien paketointi 31
5.5.2 Kokoonpanopaketti 32
5.5.3 PyPICloudin asentaminen 32
5.6 Julkaisuputkien esittely 33
6. Arviointi ja jatkokehitys 36
6.1 Lopputulos 36
6.2 Tulokset 38
6.3 Jatkokehitys. 39
7. Yhteenveto 40
Lähteet 46

1. JOHDANTO

Ohjelmistokehitys on muuttunut kahden viime vuosikymmenen aikana paljon. Aikaisemmin kehitys oli hidasta ja ohjelmistoprojektit pitkiä. Nykyään kehitys on nopeampaa ja projektit lyhytikäisempiä. Tähän on vaikuttanut ohjelmistosuunnittelun kehittyminen ja tietotekniikan halpeneminen, joka on mahdollistanut tehokkaampien prosessien käyttöönoton. [1]

Tämän diplomityön tarkoituksena on kehittää OneByte Oy:n nykyistä perinteiseen ohjelmistokehitykseen perustuvaa Python-kehitysympäristöä vastamaan paremmin nykyaikaista ohjelmistokehitystä. Tavoitteet kehitysympäristön kehityksessä ovat seuraavanlaiset:

- Automatisoinnin lisääminen. Yksitoikkoisia, virheherkkiä ja puuduttavia rutiinitoimenpiteitä pitäisi kehitysprosessista olla mahdollisimman vähän.
- Virheenjäljityksen parantaminen. Ohjelmassa ilmenneiden virheiden jäljittäminen tulisi olla helppoa ja virhelokit keskitetysti kehittäjien saatavilla.
- Nopeuden parantaminen. Kehitysprosessin tulisi olla nopea, jotta asiakkaan haluamat ominaisuudet saadaan kehitettyä ja vietyä tuotantoon nopeasti.

Kehitysympäristön kehittäminen alkoi jo muutama vuosi ennen työn varsinaista aloittamista. Aikaisemman kehityksen ratkaisut olivat samankaltaisia kuin jatkuvan toimituksen käytännössä, joka ratkaisee perinteiseen ohjelmistokehitykseen liittyviä ongelmia. Jatkuva toimitus on kyky käyttöönottaa ohjelmaan tehdyt muutokset turvallisesti ja nopeasti tuotantoon automatisointia apuna käyttäen [2].

Tässä työssä jatketaan jatkuvan toimituksen käytännön soveltamista ja perehdytään siihen sekä sen tarvitsemiin työkaluihin perusteellisemmin. Sen jälkeen pohditaan miten hyvin kehitysympäristö soveltaa jatkuvan toimituksen käytäntöä ja miten sitä voisi hyödyntää paremmin.

Luvussa 2 kerrotaan jatkuvan toimituksen periaatteet ja siihen liittyviä ratkaisuja. Lisäksi käydään lävitse jatkuvan toimitukseen liittyviä hyötyjä ja haasteita sekä sen eroavaisuudet jatkuvaan käyttöönottoon.

Luvussa 3 kerrotaan jatkuvan toimituksen työkaluista yleisesti. Samalla esitellään työkaluja, joita työssä hyödynnetään.

Luvussa 4 luvussa esitellään työn kohteeksi valikoinut Anitta laskutus- ja perintäjärjestelmä. Tämän jälkeen käydään lävitse Anittan kehitysympäristö ja siihen liittyvät ongelmat.

Luvussa 5 luvussa esitellään työn toteutusosuus. Aluksi käydään lävitse toteutukseen liittyvien työkalujen valintakriteerit. Tämän jälkeen esitellään kronologisessa järjestyksessä toteutukseen liittyvät ratkaisut.

Luvussa 6 esitetään työn lopputulos ja siihen liittyvät tulokset sekä ehdotuksia jatkokehitystä varten. Viimeisessä luvussa käydään lävitse työn yhteenveto.

2. JATKUVA TOIMITUS

Jatkuva toimitus (engl. *continuous delivery*, *CD* tai *CDE*) on jatkuvan ohjelmistosuunnittelun (engl. *continuous software engineering*) paradigmaan kuuluva käytäntö. Käsitteenä jatkuva toimitus laajentaa jatkuvan integroinnin (engl. *continuous integration* tai engl. *CI*) käytäntöä käyttöönottoautomaatiolla. Näin ollen jatkuva toimitus edellyttää jatkuvaa integrointia. [3]

Jatkuva toimitus on kyky käyttöönottaa ohjelmaan tehdyt muutokset turvallisesti ja nopeasti tuotantoon. Muutokset eivät koske ainoastaan uusia ominaisuuksia tai bugikorjauksia vaan myös konfiguraatiomuutoksia sekä kokeellisia muutoksia. Tavoitteena on tehdä käyttöönotoista helppoja, ennustettavia ja nopeasti suoritettavia rutiininomaisia toimenpiteitä. Perinteisestä ohjelmistokehityksestä poiketen käyttöönottoja voidaan suorittaa milloin vain aina tarpeen vaatiessa (engl. *on-demand*). Käyttöönotot voidaan tehdä mihinkä tahansa ympäristöön laajan mittakaavaan hajautetuista järjestelmistä sulautettuihin järjestelmiin. Tämä saavutetaan automatisoidulla julkaisuputkella (engl. *deployment pipeline*). Putki huolehtii, että koodipohja on aina käyttöönotettavassa tilassa riippumatta siitä, kuinka monta muutosta koodipohjaan tehdään päivän aikana. [2]

Viimeaikaisten tutkimusten valossa jatkuva toimitus ja käyttöönotto on kasvattanut suosiotaan ohjelmistoyritysten keskuudessa [4]. Sitä hyödynnetään pienistä organisaatiosta suuriin organisaatioihin [5].

2.1 Periaatteet

Jatkuvan toimituksen taustalla vaikuttavat periaatteet tiivistyvät kahdeksaan kohtaan, jotka David Farley ja Jez Humble esittelivät ensimmäisenä vuonna 2010 kirjassa *Continuous Delivery* [6]. Periaatteet ovat mukaillen kääntäen seuraavat:

1. Rakenna toistettava ja luotettava prosessi ohjelman julkaisuun.
2. Automatisoi lähes kaikki.
3. Pidä kaikki tieto versionhallinnassa.
4. Jos jokin on vaikeaa, tee sitä useammin.

5. Rakenna laatu sisään.
6. Valmis tarkoittaa julkaistua.
7. Jokainen on vastuussa toimitusprosessista.
8. Jatkuva kehitys.

Periaatteissa automaatio on kaiken keskiössä. Ohjelman kehitysprosessissa jokainen vaihe tulisi automatisoida siihen pisteeseen asti, jossa tarvitaan ihmisen ohjausta tai päätöksentekoa. Kehitysprosessin tulee myös olla luotettava ja toistettavissa.

Luotettavuus ja toistettavuus edellyttää, että kaikki kehitykseen liittyvä tieto, lähdekoodien lisäksi, on saatavilla versionhallintajärjestelmästä. Tällaista tietoa ovat tekniset dokumentit, erilaiset skriptit testauksesta käyttöönottoon, ohjelmakirjastot, työkaluketjut (engl. *toolchain*) ja niin edelleen. Ohjelma pitää olla mahdollista rakentaa ja käyttöönottaa käytettävissä oleviin ympäristöihin yhdellä komennolla versionhallintajärjestelmästä saatavilla tiedoilla.

Kuten Lean-ajattelussakin, jatkuvassa käyttöönotossa pyritään havaitsemaan ja korjaamaan virheet mahdollisimman aikaisessa vaiheessa. Mitä aikaisemmassa vaiheessa kehitystä virheet havaitaan, sitä helpommaksi niiden korjaaminen tulee. Lisäksi itse kehitysprosessia tulee säännöllisesti kehittää yhdessä kaikkien kehityksessä mukana olevien kanssa.

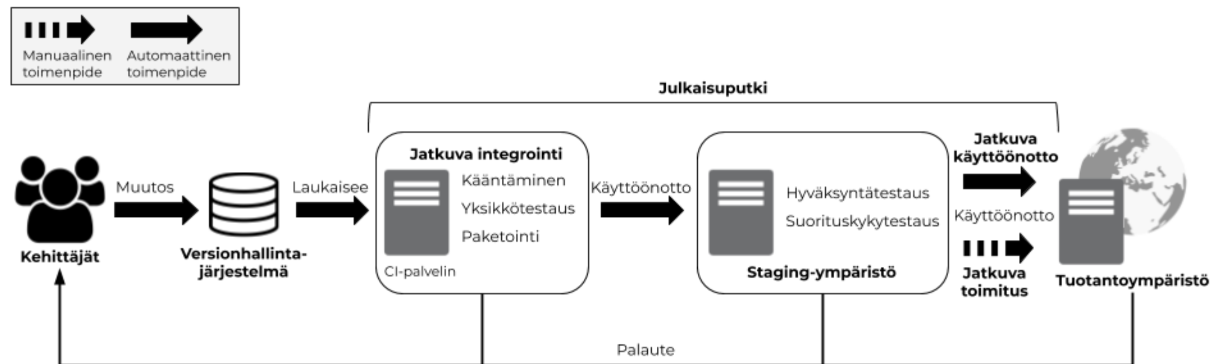
Periaatteet painottavat yhteistyötä ja säännöllistä kommunikointia kehityksessä mukana olevien sidosryhmien (kehittäjät, ylläpitäjät, johto, asiakas ja niin edelleen) välillä. Tavoitteena on, että kaikki ryhmät ovat yhdessä vastuussa ohjelman kehityksestä eikä ryhmien välistä siiloutumista tapahtuisi. Käytännössä tämä tarkoittaa, että kehitykseen liittyvä tieto on vapaasti kaikkien sidosryhmien nähtävissä yhdestä paikasta.

2.2 Perusta

Jatkuvan toimituksen perusta rakentuu versiohallintajärjestelmän ja sitä käyttävän julkaisuputken (engl. *deployment pipeline* tai engl. *continuous delivery pipeline*) ympärille (Kuva 2.1). Ohjelmaan kehityksen aikana tehdyt koodimuutokset (engl. *commits*) kulkevat aina julkaisuputken läpi ennen varsinaista julkaisua ja käyttöönottoa tuotantoon. [6]

Julkaisuputki on prosessi, jonka tarkoitus on havaita koodimuutokset, jotka aiheuttavat ongelmia ohjelman toiminnassa. Ongelmia voi ohjelman yleisen toimimattomuuden lisäksi olla suorituskykyyn, tietoturvaan ja käytettävyyteen liittyvät puutteet. Tällaiset muu-

tokset pysäyttävät julkaisuputken, estäen muutoksen julkaisun ja käyttöönoton. Tällöin kehittäjien tehtäväksi jää joko kyseisten muutosten korjaaminen tai peruminen (engl. *revert*), jonka jälkeen prosessi aloitetaan alusta. [7, 6]



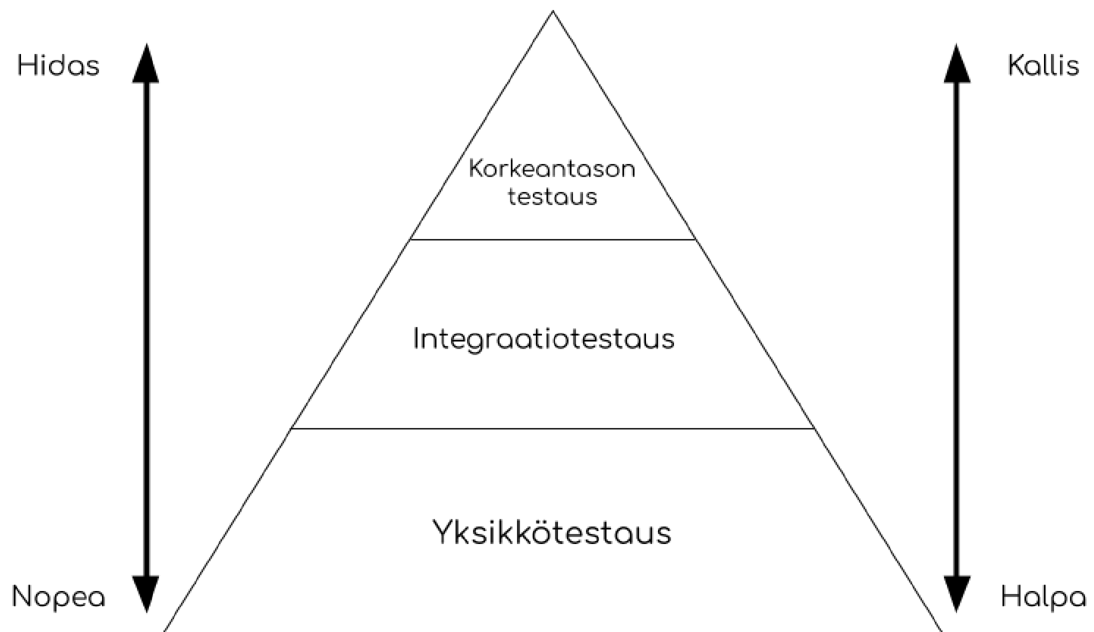
Kuva 2.1: *Kuvaus jatkuvasta toimituksesta ja -käyttöönotosta [4].*

Versionhallintajärjestelmässä olevien ohjelmaversioiden tulee olla mahdollista jäljittää ympäristöön, jossa kyseistä ohjelmaversiota ajetaan. Poikkeuksena ovat ohjelmiston lähdekoodista käännetyt binääritiedostot. Niiden tallentaminen versionhallintaan ei ole järkevää, sillä ne pitäisi muutenkin pystyä rakentamaan versionhallinnassa jo olevilla tiedoilla. Lisäksi ne vievät paljon turhaa tilaa versiohallinnassa. [6]

2.2.1 Jatkuva testaus

Jatkuvalla testauksella varmistetaan, että koodipohjaan tehdyt muutokset ovat suurella todennäköisyydellä julkaisukelpoisia ja eikä niistä löydy ongelmia aiheuttavia virheitä. Testauksen tulisi olla mahdollisimman nopea suorittaa, jotta se ei muodostuisi pullonkaulaksi ohjelmiston julkaisemiseksi useammin. [8]

Perinteistä manuaalista regressiotestausta tulisi välttää, sillä se kestää kauan ja on suhteellisen kallista suorittaa. Regressiotestauksella tarkoitetaan aiemmin testatun ohjelman uudelleentestaamista varmistaen, että ohjelmaan tehty muutos ei ole aiheuttanut virheitä ohjelmassa ennen muutosta olleeseen toiminallisuuteen [9]. Palautteen saaminen testeistä kehittäjille voi kestää viikkoja sen jälkeen, kun he ovat kirjoittaneet testattavan koodin. Manuaaliset regressiotestit ovat herkkiä inhimillisille virheille, koska ihmiset soveltuvat huonosti toisteisten tehtävien suorittamiseen. Lisäksi niissä käytettävien testausdokumentaatioiden päivittäminen ja ylläpitäminen vaatii huomattavan paljon vaivaa, kun ohjelmat kehittyvät. [8]



Kuva 2.2: *Kuvaus testauspyramidista [10].*

Testauksessa tulisi hyödyntää yksikkötestausta, integraatiotestausta sekä korkeantasontestausta. Korkeantason testauksella tarkoitetaan testausmenetelmiä, jotka testaavat ohjelman toimintaa mustalaatikkoperiaatteella. Näitä menetelmiä ovat esimerkiksi käyttöliittymätestaus ja hyväksyntätestaus. Testausmenetelmät olisi hyvä priorisoida yllä esitetyn testauspyramidin (Kuva 2.2) mukaisesti. Se on suuntaa antava kuvaus yleisesti hyväksi todetusta testausstrategiasta. Pyramidissa yksikkötestaus on painoarvoltaan suurin ja korkeantasontestaus pienin. Integraatiotestaus sijoittuu näiden väliin. [10, 11]

Yksikkötestejä tulisi käyttää testauksessa eniten. Ne ovat erittäin nopeita suorittaa ja niillä pystyy testaamaan ohjelman eri suorituspolut sekä reunaehdot kattavasti yksikkötasolla. Niiden ylläpitäminen on myös vaivatonta. Seuraavaksi eniten tulisi olla integraatiotestejä. Ne ovat hitaampia kuin yksikkötestit eivätkä anna yhtä tarkkaa tietoa virheenlaadusta kuin yksikkötestit. Niitä ei tulisi käyttää yksikkötestauksen korvaajina. Niiden ylläpitäminen on myös vaikeampaa. Korkeantasontestejä tulisi olla vähiten. Ne ovat hitaita suorittaa ja niiden kirjoittaminen on kallista, minkä takia niitä tulisi käyttää vain ohjelman eri käyttötapausten testaamiseen. Korkeantasontestit ovat herkkiä ohjelman muutoksille, mikä tekee niistä myös kalliita ylläpitää. [11]

Yksikkö- ja integraatiotestit ovat testauksen kannalta tärkeimpiä. Mikäli yksikin niistä epäonnistuu, ei ohjelmaa voida julkaista. Yksikkö- ja integraatiotestien ajoympäristön tulee olla mahdollisimman neutraali. Ympäristön ei tarvitse olla identtinen kopio tuotantoympäristöstä. Tällöin testien suoritus on nopeampaa ja halvempaa. [6]

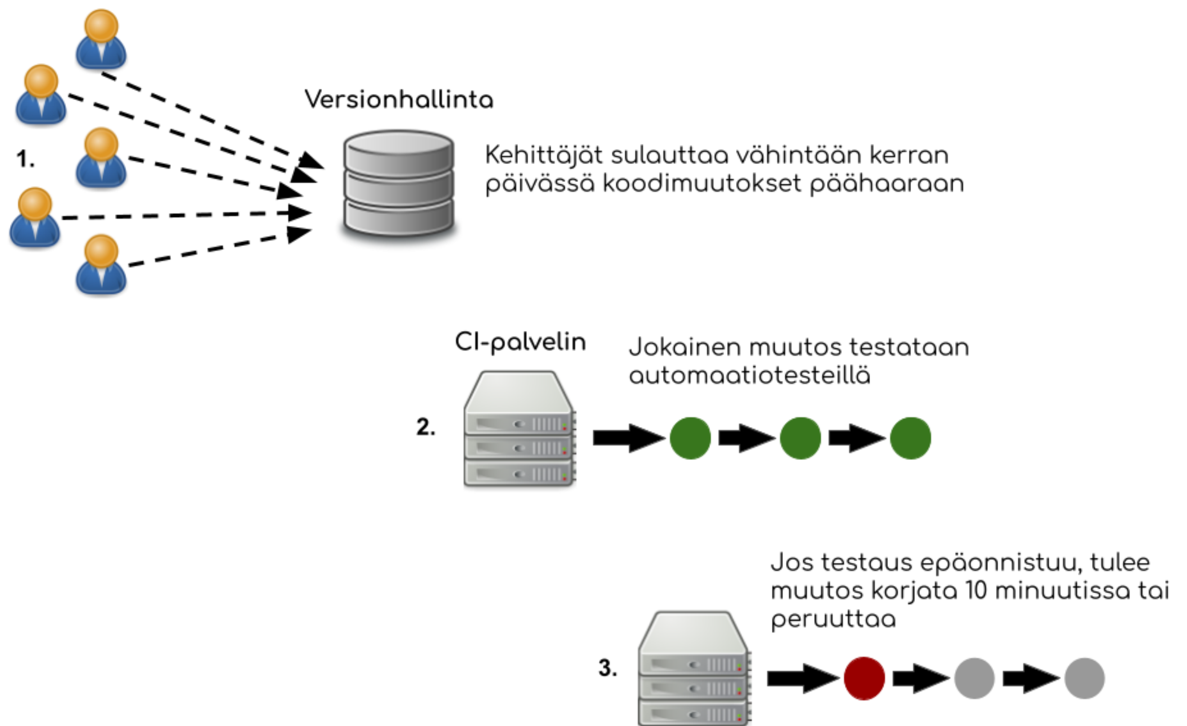
Korkeantasontestit eivät ole testauksen kannalta yhtä kriittisiä. Osa testeistä voi epäonnistua, mutta se ei välttämättä estä ohjelman julkaisua, jos kyseessä on esimerkiksi vain suorituskykyä heikentävä ongelma. Korkeantasontestien ajoympäristön tulee kuitenkin olla mahdollisimman yhdenmukainen tuotantoympäristön kanssa. [6]

Testausta tulisi kehittää niin, että virheet löydetään mahdollisimman aikaisessa vaiheessa testausta. Mikäli virhe huomataan esimerkiksi vasta tutkivassa testauksessa, tulisi automaatiotestejä kehittää havaitsemaan se. Mikäli taas virhe havaitaan integraatiotesteissä, tulisi yksikkötestejä kehittää havaitsemaan se. Näin jatkossa sama virhe huomataan aikaisemmassa vaiheessa testausta. [10]

2.2.2 Jatkuva integrointi

Jatkuva integrointi on ohjelmistotuotannon käytäntö, jossa ohjelmiston koodipohjaan tehtävät muutokset (engl. *commits*) sulautetaan (engl. *merge*) rutiininomaisesti vähintään kerran päivässä versionhallinnan päähaaraan (engl. *trunk* tai engl. *master*). Ideaalitilanteessa sulautuksia tehdään useita kertoja päivässä. Kaikkien kehittäjien tulisi myös vetää (engl. *pull*) päivittäin päähaaraan tehdyt muutokset ja sulauttaa ne tekeillä olevaan työhön. [12]

Jokainen versiohallintaan tehty muutos laukaisee sarjan nopeita automaatiotestejä, joilla havaitaan muutosten mahdollisesti aiheuttamat ongelmat. Kehittäjien tulee heti korjata ongelmalliset muutokset tai kumota (engl. *revert*) ne, mikäli niitä ei saada ajoissa korjattua. Jatkuvan integroinnin tuloksena ohjelmiston koodipohja on aina ehjässä tilassa, jolloin siitä saadaan tarvittaessa koontiversioita (engl. *builds*) ja paketteja (engl. *packages*), jotka voidaan milloin vain käyttöönottaa ja julkaista. [6]



Kuva 2.3: *Kuvaus jatkuva integroinnin toiminnasta [13].*

Perusajatuksena jatkuvassa integroinnissa on estää koodipohjan eriytymistä kehityksen aikana. Tämä tehdään pitämällä päähaaraan kerralla sulautettavan koodin määrä mahdollisimman pienenä. Kun sulautettavan koodin määrä on pieni, sulautuksista syntyvät sulautuskonfliktit (engl. *merge conflicts*) ovat pienempiä ja helpompia ratkaista. Samalla koodipohjan refaktoroinnista tulee helpompaa, kun kehittäjät ovat koko ajan tietoisia koodipohjan todellisesta tilasta. [13]

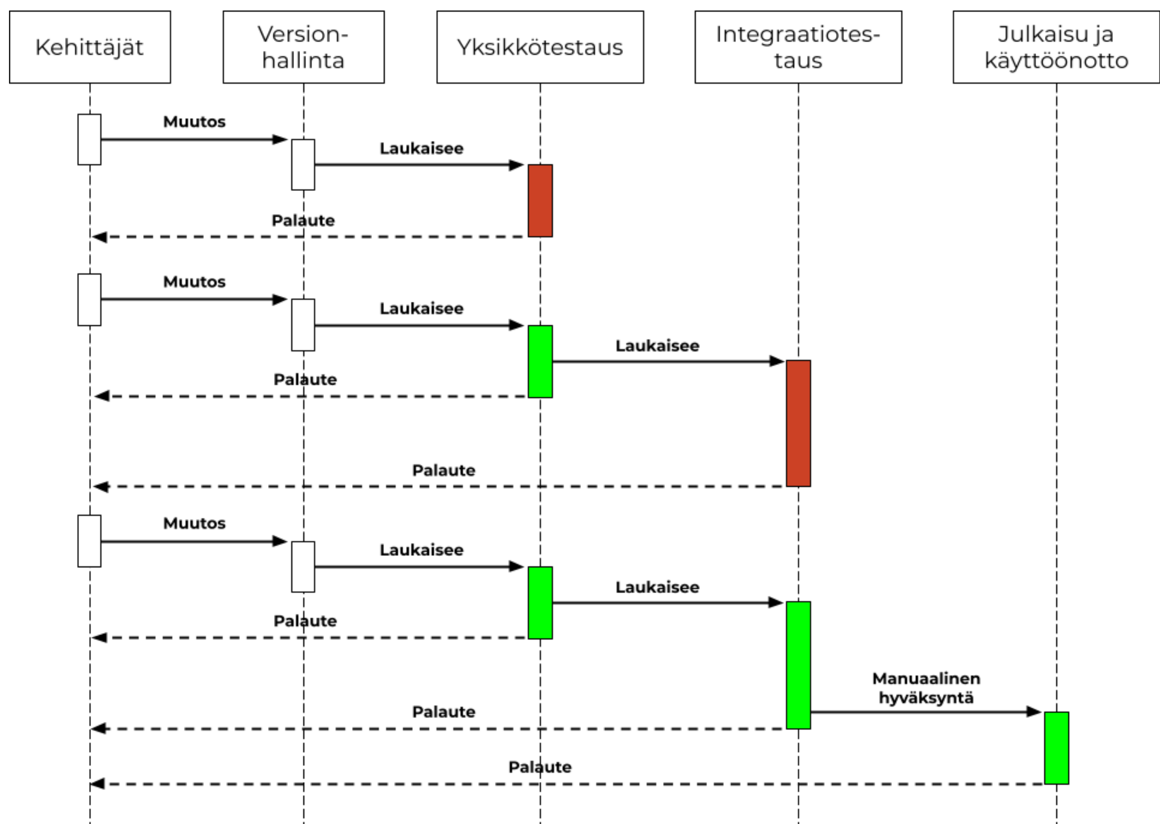
Automaatiotestien suoritusaikojen tulisi jatkuvassa integroinnissa olla alle kymmenen minuuttia. Mikäli suoritusajat ovat tätä suurempia, testausta tulisi kehittää nopeammaksi. [13]

Kehityksen aikana tulisi välttää päähaaran haarauttamista (engl. *branching*), jota käytetään ominaisuushaarautus-kehitysmallissa (engl. *feature branching*). Haarauttamisen seurauksena päähaaran koodipohja eriytyy ja tekee muutoksien sulauttamisesta takaisin päähaaraan vaivalloisempaa. Tämä vähentää kehittäjien halua koodipohjan refaktorointiin. Haarauttamista ei kuitenkaan kokonaan kielletä, mikäli haarojen elinikä jää lyhyeksi ja niiden sisältämät muutokset ovat pieniä. [14, 15]

2.2.3 Julkaisuputki

Julkaisuputki on keskeisin osa jatkuvaa toimitusta. Se on prosessi, jolla ohjelmaan tehdyt koodimuutokset käännetään, testataan, julkaistaan ja käyttöönotetaan tuotantoon. Se koostuu monesta peräkkäisestä vaiheesta (engl. *stages*) ja niille määritetyistä tehtävistä (engl. *tasks*). Tehtävät vastaavat yhtä toimenpidettä kuten esimerkiksi yksikkötestien ajamista. Tehtävät voivat olla automaattisia tai manuaalisia. Automaattiset tehtävät suoritetaan ilman käyttäjän hyväksyntää, kun taas manuaaliset tehtävät vaativat aina käyttäjän hyväksynnän. [16]

Julkaisuputki käynnistyy, kun versionhallintajärjestelmään tehdään koodimuutos. Putken suoritus aloitetaan ensimmäisen vaiheen ensimmäisestä tehtävästä. Vaiheelle määritettyjä tehtäviä voidaan myös suorittaa rinnakkain, jolloin vaiheen suoritusaikaa saadaan lyhennettyä. Kun vaiheen kaikki tehtävät on suoritettu, siirrytään seuraavaan vaiheeseen. Näin jatketaan, kunnes päästään putken viimeisen vaiheeseen, jonka jälkeen putken suoritus on valmis. Mikäli yksikin vaiheen tehtävistä epäonnistuu, keskeytyy julkaisuputken suoritus. [16]

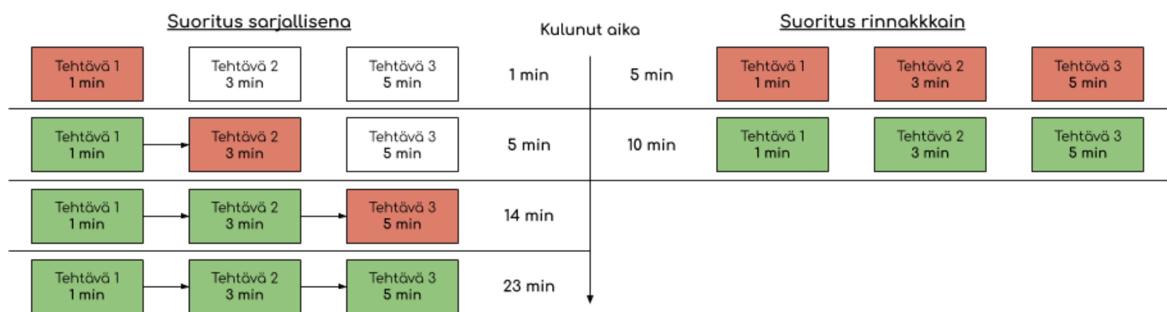


Kuva 2.4: *Kuvaus yksinkertaisen kehitysputken suorituksesta [17].*

Jokaisesta julkaisuputken tehtävästä lähtee kehittäjille palaute, joka antaa tietoa tehtävien tuloksista luoden samalla kokonaiskuvaa putken tilasta. Palautteet helpottavat virheiden diagnosointia ja korjaamista. Kun muutos on edennyt putken loppuun, on se suurella todennäköisyydellä tuotantoon kelpaava. Tämän jälkeen se voidaan julkaista ja käyttöönottaa tuotantoon manuaalisesti napin painalluksella. Kuvassa 2.4 on kuvaus yksinkertaisen julkaisuputken suorituksesta.

Hyvälle julkaisuputkelle on ominaista lyhyt läpimenoaika. Nopeat tehtävät, kuten koodin staattinen analysointi (engl. *code linting*) ja yksikkötestaus, tulisi suorittaa putken alussa. Näin putken mahdollinen epäonnistuminen tapahtuu aikaisessa vaiheessa. Hitaat tehtävät, kuten hyväksymistestaus ja manuaaliset tarkistukset, tulisi suorittaa putken lopussa. Tässä vaiheessa koodimuutos on jo yleensä julkaisukelpoinen eikä sisällä kriittisiä virheitä. [18]

Eri vaiheiden tehtävät tulisi suorittaa mahdollisuuksien mukaan rinnakkain, jolloin läpimenoaikaa saadaan pienemmäksi [18]. Tämä on havainnollistettu vertailun avulla alla olevassa kuvassa.



Kuva 2.5: *Kuvaus sarjallisen ja rinnakaisen suorituksen eroista [18].*

Vertailussa oletetaan, että vaiheen jokainen kolmesta tehtävästä epäonnistuu ensimmäisellä suorituskerralla. Suoritettaessa tehtävät sarjallisena tarvitaan neljä yrityskertaa muutoksen läpiviemiseen, kun taas rinnakkain suoritettuna vain kaksi. Rinnakkain suoritettuna vaiheen läpimenoaika on yli kaksi kertaa vähemmän kuin sarjallisena suoritettuna. [18]

2.3 Hyödyt

Vaikka käytäntönä jatkuva toimitus on melko tuore ja vähän tutkittu, on sillä havaittu olevan ohjelmistokehitykseen myönteisiä vaikutuksia. Leppäsen ja muiden [5] toteuttamassa empiirisessä tutkimuksessa tehtiin seuraavanlaisia havaintoja:

- Palautteen saaminen kehitysprosessista ja loppukäyttäjiltä nopeutui.

- Julkaisuja pystyttiin tekemään nopeammin ja tiheämmin.
- Tuottavuus parantui.
- Ohjelman laatu parantui.
- Asiakastyytyväisyys lisääntyi.
- Manuaalinen työ vähentyi.

Kun palautteen saaminen kehitysprosessista ja loppukäyttäjiltä oli nopeampaa, kehittäjät näkivät paremmin onko kehitteillä oleva ominaisuus sellainen kuin on haluttu. Päätöksiä kehityksen jatkamisesta ja lopettamisesta oli näin helpompi tehdä. Nopeampi palaute kehitysprosessista vahvisti myös kehittäjien tunnetta omasta suorituksesta ja kasvatti heidän työmotivaatiotaan. Tämän on todettu myös lisäävän kehittäjien tyytyväisyyttä ja pienentävän loppuunpalamisen riskiä. [5, 2]

Tihentynyt julkaisutahti vähensi radikaalisti uusien ominaisuuksien ja tuotteiden kehitykseen kuluvaa aikaa (engl. *time-to-market*). Loppukäyttäjät ja muut sidosryhmät olivat aiempaa tietoisempia kehityksen tilasta, koska konkreettisia tuloksia ohjelman kehittymisestä saatiin aiempaa useammin nähtäville. Tämä vaikutti asiakastyytyväisyyteen positiivisesti. [5, 19]

Jatkuvassa käyttöönotossa ja toimituksessa korostetaan testausautomaatiota sekä suppeampia ohjelmajulkaisuja. Näiden todettiin parantavan ohjelmakoodin laatua sekä ohjelman toiminnallisuutta. Automatisoinnin myös todettiin säästävän aikaa ja vaivaa, suoraviivaistamalla julkaisuprosessia ja vähentämällä manuaalisen työn määrää. Tällöin myös tuottavuus parantui. [5, 19]

Jatkuvassa toimituksessa poistuu tarve perinteisessä ohjelmistosuunnittelussa käytetyille integrointi- ja testausvaiheille, jotka ovat manuaalisia ja voivat kestää useita viikkoja tai jopa kuukausia. Kehittäjille jää näin enemmän aikaa keskittyä esimerkiksi käyttäjätutkimukselle ja korkeantason testaamiselle kuten tutkivaan-, käytettävyys-, suorituskyky- ja turvallisuustestaukseen. [2]

2.4 Haasteet

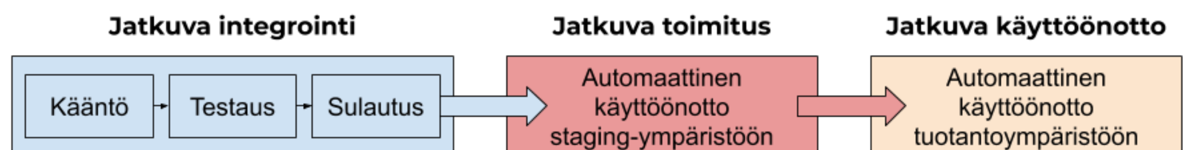
Kuten missä tahansa ohjelmistokehityksen käytännössä, myös jatkuvaan toimitukseen siirtymisessä voi olla haasteita. Leppänen et al. [5] toteuttamassa empiirisessä tutkimuksessa havaittiin useita tekijöitä, jotka hidastivat käytäntöön siirtymistä.

Jatkuva toimitus edellyttää kattavaa, nopeaa ja automatisoitua testausta toimiakseen kunnolla. Tutkimuksessa havaittiin, että automatisoidun testauksen puute on merkittävä hidaste jatkuvaan käyttöönottoon siirryttäessä. Automaatiotestausta ei välttämättä pystytty tekemään kunnolla monesta eri syystä. Vanhojen ohjelmien (engl. *legacy sofware*) kohdalla automaatiotestejä ei välttämättä ollut saatavilla eikä niitä ollut mahdollista niihin tehdä. Ohjelmat saattoivat myös olla toteutukseltaan niin monimutkaisia, että testaus täytyi suorittaa osittain manuaalisesti. Erityisesti käyttöliittymien testaus koettiin haastavaksi toteuttaa automatisoidusti. Toisaalta ohjelmille saattoi olla olemassa kattavat automaatiotestit, mutta niiden suorittaminen kesti useita tunteja, jolloin testauksesta muodostui pullonkaula nopealle julkaisulle. [5]

Yrityksen sisäisen muutosvastarinnan lisäksi jatkuvaan toimitukseen siirtyminen voi aiheuttaa muutosvastarintaa myös asiakkaissakin. Tutkimuksessa havaittiin, että asiakkaat eivät välttämättä olleet halukkaita siirtymään jatkuvaan toimitukseen. Haasteena on erityisesti toimialakohtaiset käytännöt ja rajoitukset, jotka voivat olla ristiriidassa jatkuvan toimituksen käytäntöjen kanssa. [5]

2.5 Erot jatkuvaan käyttöönottoon

Käsitteenä jatkuva toimitus saatetaan helposti sekoittaa jatkuvaan käyttöönottoon [20]. Se on käytäntönä hyvin samankaltainen jatkuvan toimituksen kanssa, mutta tuotantovalmiin ohjelman toimittaminen tuotantoympäristöön on täysin automaattinen toimenpide. Toisin sanoen, jatkuvassa käyttöönotossa keskitytään julkaisemaan jokainen ohjelmaan tehty muutos ja toimittamaan se tuotantoon automaattisesti, kun taas jatkuvassa toimituksessa oleellisempaa on pitää ohjelma käyttöönottokykyisenä. [3]



Kuva 2.6: *Kuvaus jatkuvan integroinnin, toimituksen ja käyttöönoton eroista [21].*

Jatkuva käyttöönotto edellyttää jatkuvaa toimitusta ja on näin ollen jatkoa jatkuvalle toimitukselle. Toisinpäin ajateltuna tämä ei kuitenkaan ole totta [22]. Molemmat käytännöt edellyttävät jatkuvaa integrointia. Käytäntöjen väliset riippuvuudet on havainnollistettu kuvassa 2.6.

3. JATKUVAN TOIMITUKSEN TYÖKALUT

Toimiakseen jatkuva toimitus vaatii työkaluja. Tavallisen versionhallintajärjestelmän lisäksi tarvitaan työkaluja muun muassa julkaisuputkien vaatimien prosessien toteuttamiseen ja monitorointiin. Työkaluja on paljon ja niitä voidaan kategorisoida monella eri tavalla. [1]

Eroja työkalujen välillä löytyy niin hinnassa, asennustyypissä kuin ylläpidossa. Tavallisesti työkaluista on tarjolla kahdentyyppistä asennusta: paikallinen (engl. *on-premise*) tai palvelupohjainen eli SaaS (*Software As a Service*). Paikallisessa vaihtoehdossa työkalu asennetaan omalle palvelimelle ja sitä ylläpidetään itse. Palvelupohjaisessa vaihtoehdossa työkalua ei tarvitse itse asentaa ja sen ylläpidosta huolehtii erillinen palveluntarjoaja. [23]

Paikallisesti asennetun työkalun etuna on joustavuus ja alhaiset käyttökustannukset. Ilmaisissa ja avoimen lähdekoodin työkaluissa hinta koostuu vain palvelin- ja ylläpitokustannuksista. Tietojenkäsittely tapahtuu yrityksen omilla palvelimilla, jolloin yrityksellä on mahdollisuus tarkkaan määritellä mihin tietoja tallennetaan ja käytetään. Tämä voi olla ratkaiseva tekijä, jos yrityksen tietoturvapolitiikka ei salli tietojen tallentamista kolmannen osapuolen tarjoamiin palveluihin. Heikkoutena paikallisissa asennuksissa voi olla todellisen hinnan hämärtyminen, sillä etenkin ylläpitokustannuksia saattaa olla vaikeaa arvioida ennalta. Avoimen lähdekoodin työkaluissa tuki on yleensä täysin kehittäjäyhteisöjen varassa, mikä ei aina takaa nopeaa avunsaantia työkalun käytössä ilmenneisiin ongelmiin. [23]

Palvelupohjaisen työkalun etuna on ylläpidon helppous ja käyttökustannuksien ennakoitavuus. Hinta voi määräytyä esimerkiksi käytön tai käyttäjien lukumäärän perusteella, jolloin todellisen hinnan laskeminen on helppoa. Ylläpitokustannuksia ei ole, sillä palveluntarjoaja huolehtii ylläpidosta ja tarjoaa tuen työkalulle. Heikkoutena on tiedonkäsittelyn hämärtyminen. Palveluntarjoajat eivät aina kerro kuinka, miten ja mihin palveluun tallennettuja tietoja käsitellään. Tämä voi herättää epäilyksiä palveluntarjoajan luotettavuudesta ja kyvystä ylläpitää palvelua. [23]

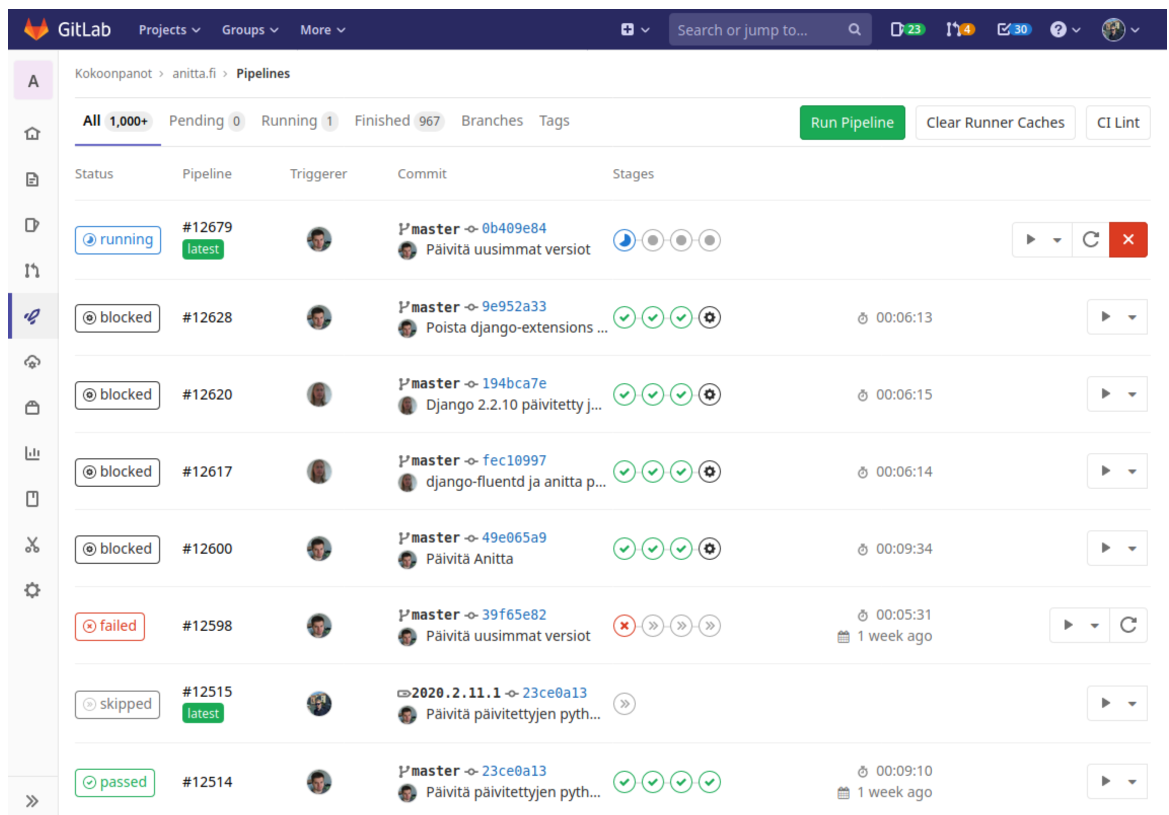
Tässä luvussa käsitellään vain tämän työn kannalta oleellisimpia työkaluja, jotka on jaettu kahteen kategoriaan itse jatkuvan toimituksen ja -käyttöönoton työkaluihin sekä monitoroinnin työkaluihin. Jokaisesta työkalusta on saatavilla sekä paikallinen että palvelupohjainen asennus.

3.1 Jatkuva toimitus ja -käyttöönotto

Jatkuvan toimituksen- ja käyttöönoton työkaluja käytetään kyseisten käytäntöjen päätyökaluina. Ne tarjoavat kokonaisvaltaisen ratkaisun julkaisuputkien rakentamiseen ja ylläpitoon. [24]

3.1.1 GitLab CI/CD

GitLab CI/CD on GitLab Inc yrityksen vuonna 2012 kehittämä työkalu jatkuvan integraation, -toimituksen ja -käyttöönoton soveltamiseen ohjelmistoprojekteissa ja on osa GitLab versionhallintajärjestelmää. Käyttöönottaminen vaatii, että ohjelmistoprojektit käyttävät GitLab-versionhallintajärjestelmää lähdekoodien tietovarastona (*repository*). [25, 26]



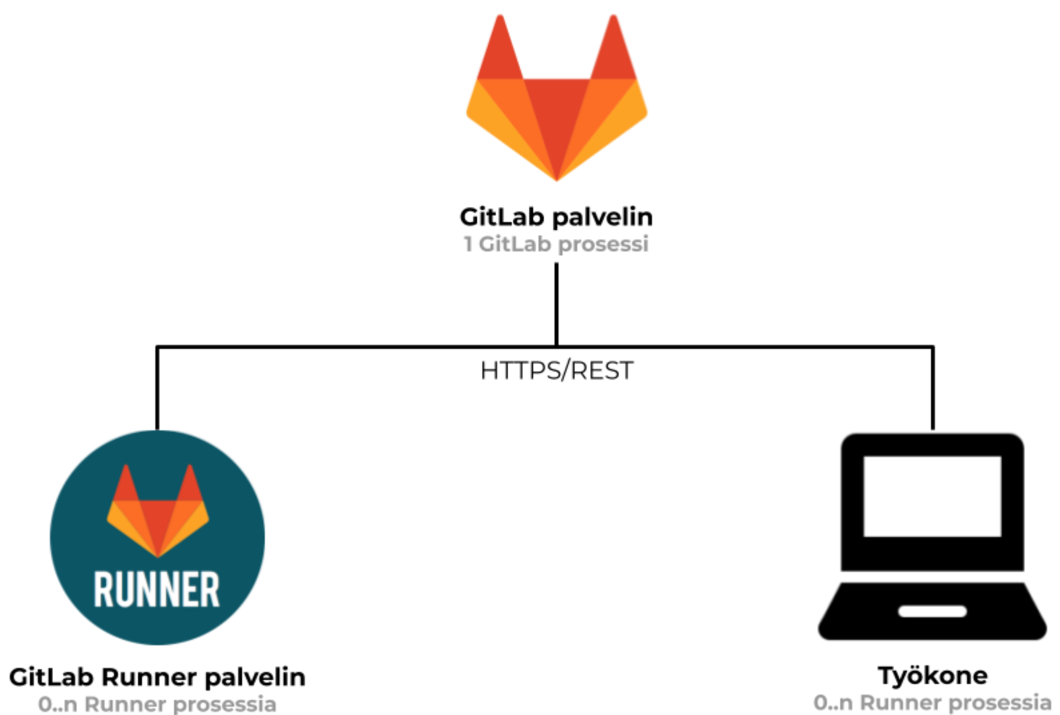
***Kuva 3.1:** Kuvakaappaus GitLab CI/CD web-käyttöliittymästä.*

GitLab CI/CD on ominaisuudet vaihtelevat käytettävän GitLab-version mukaan. Ilmainen avoimeen lähdekoodiin perustuva Community Edition versio tarjoaa kattavat ominaisuudet niin yksinkertaisten kuin monimutkaistenkin julkaisuputkien rakentamiseen. Maksullinen Enterprise Edition yritysversio tarjoaa enemmän isoihin ohjelmistoprojekteihin suunniteltuja ominaisuuksia kuten keskitetyn LDAP-käyttäjienhallinnan. GitLab CI/CD

tarjoaa selkeän web-käyttöliittymän (Kuva 3.1), jonka kautta julkaisuputkia voidaan hallita sekä niiden tilaa seurata aivan tehtävien suoritukseen asti. Forresterin tekemän tutkimuksen mukaan GitLab CI/CD oli vuonna 2017 selvästi paras työkalu jatkuvaan integrointiin. [27, 28]

GitLab CI/CD koostuu (Kuva 3.2) kahdesta osasta, GitLab pääpalvelimesta (engl. *coordinator*) sekä yhdestä tai useammasta GitLab Runner suorittajista (engl. *executor*). Pääpalvelimen tehtävänä on koordinoida julkaisuputkelle määritettyjä tehtäviä suorittajille. Suorittajat nimensä mukaisesti suorittavat annettuja tehtäviä sekä palauttavat tehtävän tilatiedot ja tulosteet pääpalvelimelle. [29]

Tehtävienvälitys toimii veto-tyyppisesti (engl. *pull*). Suorittajat pollaavat (engl. *polling*) pääpalvelimen REST-rajapintaa kysyäkseen suoritettavia tehtäviä.



Kuva 3.2: *Kuvaus GitLab CI/CD arkkitehtuurista [29].*

GitLab Runner on kirjoitettu Go-kielellä, mikä mahdollistaa sen asentamisen kaikille Go-kieltä tukeville käyttöjärjestelmille, kuten Linuxille tai Windowsille [30]. Tyypillisesti sitä ajetaan Docker-kontissa tai omana palvelininstanssina. GitLab Runneria on myös mahdollista ajaa kehittäjien omilta työkoneilta. Kirjoitushetkellä ominaisuus on kuitenkin vielä keskeneräinen [31].

3.1.2 Jenkins

Jenkins (aikaisemmin Hudson) on automaatiopalvelin jatkuvan integraation, -toimituksen ja -käyttöönoton soveltamiseen ohjelmistoprojekteissa. Sen kehitti vuonna 2004 Javalla Kohsuke Kawaguchi työskennellessään Sun-yrityksessä. [32]

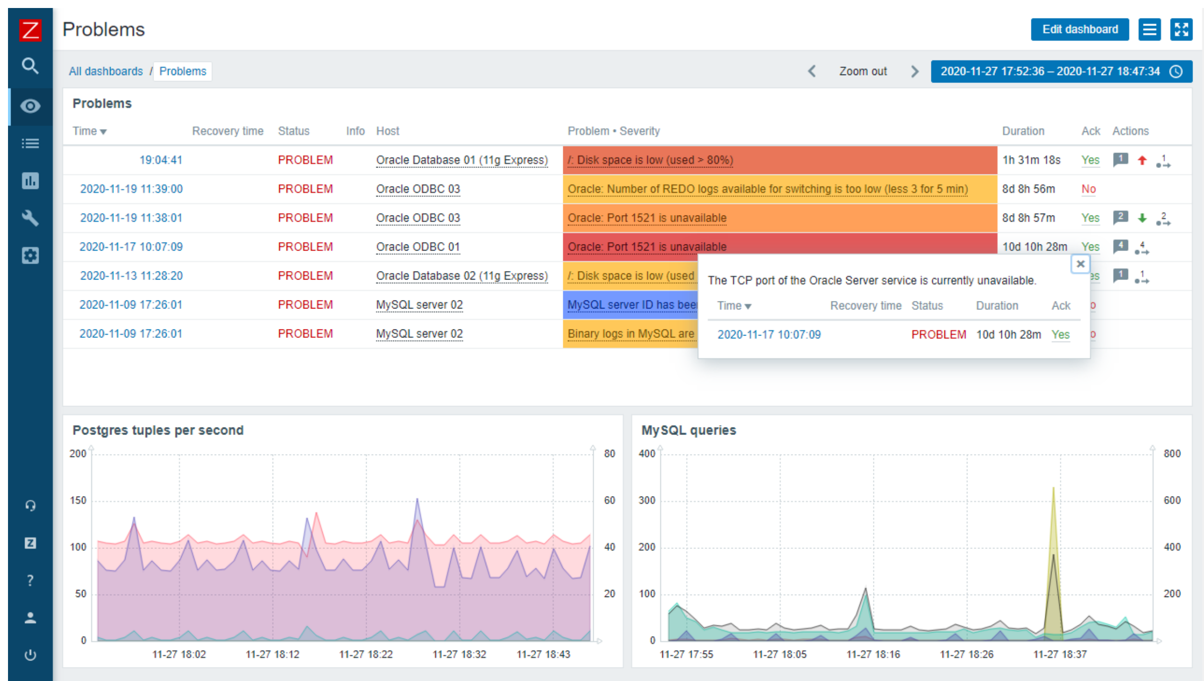
Nykyään Jenkins on johtava ilmainen avoimen lähdekoodin automaatiopalvelin, joka tukee noin 1600 laajennuksella (engl. *plugins*) kaikenlaisten kehitystehtävien automatisointia. Jenkins tarjoaa yksinkertaisen tavan luoda jatkuvan integroinnin ja -toimituksen ympäristön melkein mille tahansa ohjelmointikielen ja lähdekoodien tietovaraston yhdistelmälle. [32]

3.2 Monitorointi

Monitorointityökalujen avulla kerätään ja analysoidaan metriikoita järjestelmäinfrastruktuurin eri osista. Kerättyä tietoa voidaan käyttää sovelluksien ja palvelimien suorituskyvyn mittaamiseen ja parantamiseen. Lisäksi tietoa voidaan käyttää ongelmatilanteiden selvittämiseen ja ennaltaehkäisemiseen. [24]

3.2.1 Zabbix

Zabbix on yritystason avoimeen lähdekoodiin perustuva hajautettu monitorointijärjestelmä. Sen avulla voidaan valvoa useita verkon parametreja sekä palvelimien, virtuaalikoneiden, sovellusten, palveluiden, tietokantojen, verkkosivustojen, pilvien ja muun terveyttä ja eheyttä. Zabbix käyttää joustavaa ilmoitusmekanismia, jonka avulla käyttäjät voivat määrittä havaittujen ongelmien hälytykset käytännössä mihin tahansa mediaan kuten sähköpostiin. Tämä mahdollistaa nopean reagoinnin ongelmiin. Lisäksi Zabbix tarjoaa erinomaiset raportointi- ja tietojen visualisointiominaisuudet tallennetun datan perusteella. [33]

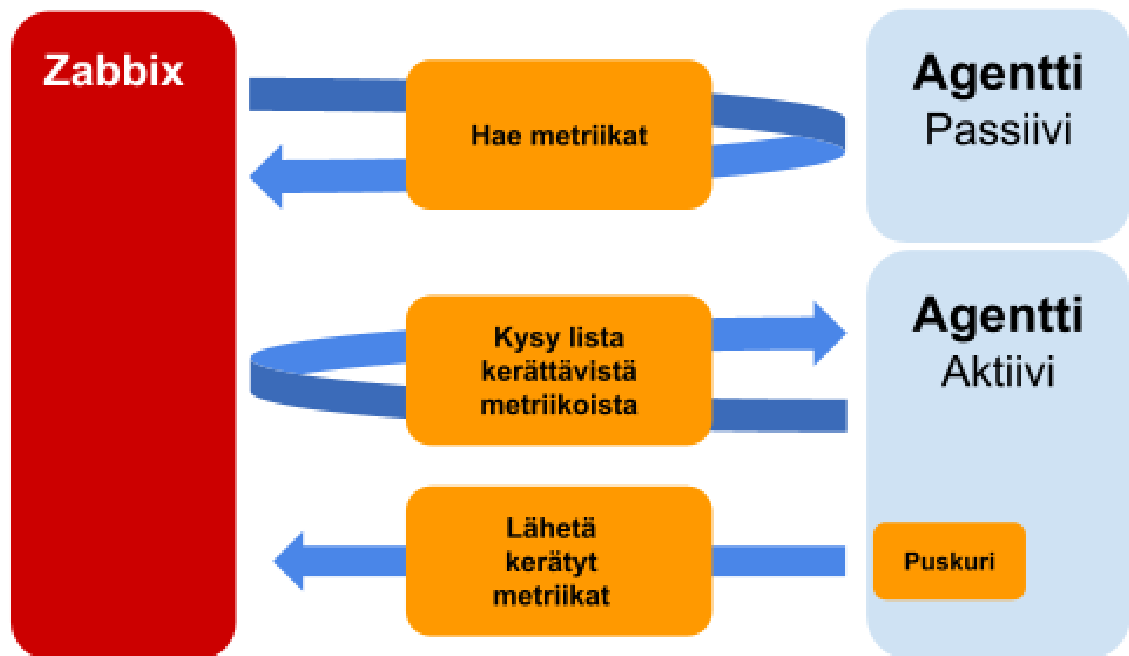


Kuva 3.3: *Kuvakaappaus Zabbixin web-käyttöliittymästä [34].*

Zabbix rakentuu seuraavista osista: ydin, käyttöliittymä, agentti. Osien tehtävät jakautuvat seuraavanlaisesti:

- **Ydin**: Vastaanottaa, prosessoi ja tallentaa agenteilta tulevat metriikat. Suorittaa monitorointia ja lähettää ilmoituksia havaituista ongelmista.
- **Käyttöliittymä**: Järjestelmän web-käyttöliittymä (Kuva 3.3). Tarjoaa muun muassa visualisoinnin kerätylle datalle.
- **Agentti**: Ohjelma, joka asennetaan monitoroitavalle palvelimelle. Kerää metriikat ja lähettää ne verkon yli ytimelle.

Zabbix-agentit voivat välittää metriikoita ytimelle joko aktiivisessa tai passiivisessa tilassa. Passiivisessa tilassa ydin pollaa agenteilta haluttuja metriikoita. Aktiivisessa tilassa sen sijaan agentit pollaavat ytimeltä listan kerättävistä metriikoista ja lähettävät sen mukaiset metriikat ytimelle. Merkittävin ero tilojen välillä on yhteyden muodostuksessa. Kuvassa 3.4 on havainnollistettu kyseisten tilojen erot. [35]



Kuva 3.4: *Kuvaus aktiivisen vs. passiivisen agentin eroista [36].*

Passiivisessa tilassa yhteys muodostetaan ytimestä agentille, mikä saattaa vaatia muutoksia monitoroitavan palvelimen palomuuriin ja verkkoympäristöön. Aktiivisessa tilassa yhteys muodostetaan agentilta ytimeen, jolloin muutoksia ei tarvitse tehdä. [35]

3.2.2 Sentry

Sentry on avoimeen lähdekoodiin perustuva sovellustason virheidenseuranta- ja jäljitystyökalu. Sen avulla kehittäjät voivat diagnosoida, korjata ja optimoida sovelluksen suorituskykyä. Sentryn tarjoaa muun muassa ohjelmavirheiden pinotiedot (engl. *stacktrace*), reaaliaikaisen virheiden esiintyvyyden seurannan ja web-pohjaisen käyttöliittymän (Kuva 3.5), jonka kautta tietoja voidaan suodattaa ja kuvantaa. [37]

Kuva 3.5: *Kuvankaappaus Sentryn web-käyttöliittymästä.*

Senty tukee yleisimpiä ohjelmointikieliä Pythonia, Rubya ja Javaa. Ohjelmointikielien lisäksi Sentrystä löytyy valmiit integraatiorajapinnat useille ohjelmointikehyksille (engl. *sofware framework*) kuten Django ja Spring. [38]

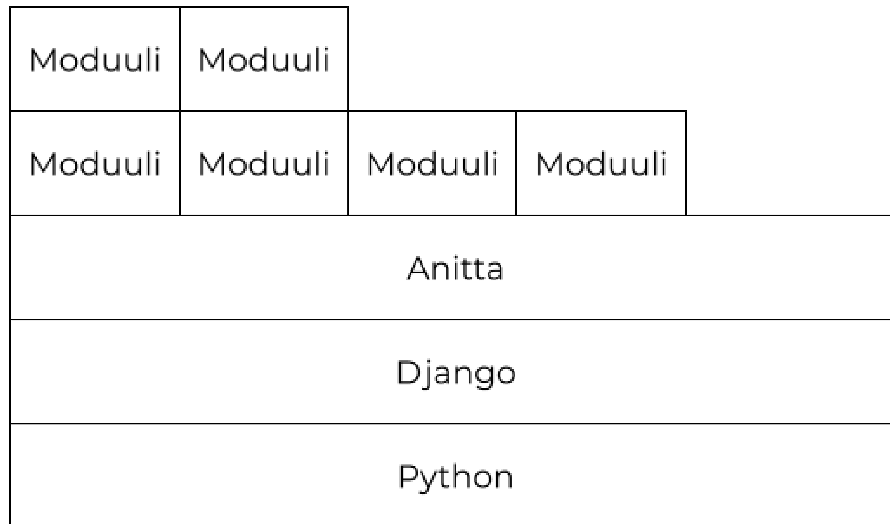
4. TYÖN KOHDE

Työn toteutuksen kohteeksi valittiin alunperin Pispalan Insinööritoimisto Oy:n vuonna 2015 kehittämä ja nykyisin OneByte Oy:n ylläpitämä web-pohjainen laskutus- ja perintäjärjestelmä Anitta. Sen suurin asiakasryhmä on Suomessa toimivat perintäalan yritykset. Anittaa voidaan käyttää tavallisten yrityksien laskutukseen, laskujenseurantaan ja maksumuistutuksien lähettämiseen. Anitta tukee yleisimpiä käytössä olevia laskutusmuotoja kuten paperi-, sähköposti- ja verkkolaskuja. Perintään Anittasta löytyy muun muassa integraatio Suomen oikeuslaitoksiin oikeudellista perintää varten. Anittasta löytyy myös kattava integraatiorajapinta, jonka avulla integrointi muiden palveluiden kanssa on helppoa ja vaivatonta.

Tässä luvussa esitellään lyhyesti Anitta ja sen kehitysympäristön lähtötilanne ennen diplomityön aloittamista. Lisäksi käydään lävitse silloinen kehitysprosessi ja siihen liittyvät ongelmat, joita tässä diplomityössä pyritään korjaamaan.

4.1 Esittely

Anitta on kirjoitettu Python 3.8:lla ja käyttää avoimen lähdekoodin Django web -sovelluskehystä (engl. *web framework*) alustanaan (engl. *platform*). Django perustuu MVC (engl. *Model-View-Controller*) ohjelmistoarkkitehtuuriin ja tarjoaa valmiit työkalut ja ratkaisut web-pohjaisten Python-sovelluksien rakentamiseen. [39, 40]



Kuva 4.1: *Korkeantason kuvaus Anittan rakenteesta.*

Korkealla tasolla Anitta koostuu Django-pääprojektista (engl. *Django project*) ja siihen liitetyistä moduuleista, jotka ovat Django-sovelluksia (engl. *Django application*). Anittan moduulit ovat riippuvaisia Anittasta ja Anitta on riippuvainen Djangosta. Tämä on esitetty kuvassa 4.1. Rakenteeltaan Anitta on modulaarinen ja sen ominaisuuksia voidaan lisätä ja laajentaa asiakkaiden tarpeiden mukaan Anitta-moduuleilla.

4.2 Kehitysympäristö

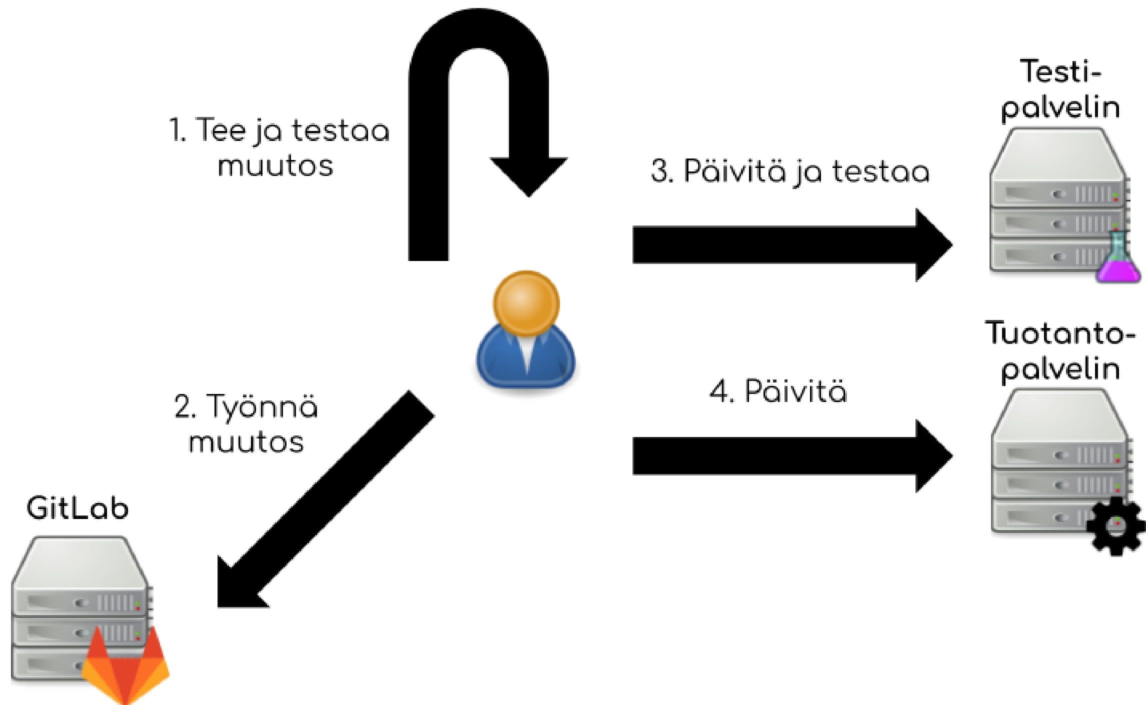
Ennen diplomityön aloittamista, Anittan koodikanta oli hyvin pieni ja kehityksestä vastasi vain kolmen hengen kehitystiimi. Tiimi koostui kahdesta kehittäjästä ja yhdestä ylläpitäjästä, joka oli diplomityön kirjoittaja. Ylläpitäjä asensi Anitta-palvelimit ja varmisti, että ne ovat aina toimintakunnossa. Kehittäjät puolestaan kehittivät ja vastasivat Anittan toiminnasta.

Kehitysympäristö koostui versionhallintajärjestelmästä, kehittäjien työasemista ja tuotanto- sekä testipalvelimista. Sekä kehittäjien työasemissa että tuotanto- ja testipalvelimissa käytettiin Linux-pohjaisia käyttöjärjestelmiä. Lähdekoodien tietovarastona (engl. *repository*) käytettiin GitLab-versionhallintajärjestelmän Community Edition -versiota.

4.2.1 Kehitysprosessi

Koodimuutoksien hallintaan käytettiin yksinkertaista haarautusmallia (engl. *feature branching*) [41]. Mallissa jokaiselle kehitettävälle omaisuudelle luodaan päähaarasta oma haa-

ransa, johon ominaisuuden vaatimat muutokset viedään (engl. *commit*). Kun ominaisuus on valmis ja testattu, tehdään siitä sulautuspyyntö (engl. *merge request*) päähaaraan. Pyyntö tehdään toiselle kehittäjälle, joka tarkistaa muutoksien toimivuuden ja hyväksyy ne, jonka jälkeen haara sulautetaan (engl. *merge*) takaisin päähaaraan.



Kuva 4.2: *Kuvaus aikaisemmasta kehitysprosessista.*

Kehitysprosessi oli perinteisen ohjelmistokehityksen kaltainen. Kaikki Anittaan ja sen moduuleihin tehdyt muutokset testattiin manuaalisesti ensin kehittäjän työasemalla. Sen jälkeen ne työnnettiin versionhallintaan ja käyttöönotettiin testipalvelimelle, jossa ne testattiin tarkemmin. Lopuksi muutokset käyttöönotettiin tuotantopalvelimelle. Jokaista testaus- ja tuotantokäyttöönottoa varten kehittäjä kirjautui SSH-yhteydellä palvelimelle, suoritti päivityskomennot ja käynnisti Anittan uudelleen. Tuotantokäyttöönoton jälkeen muutoksien toimivuus varmistettiin vielä testaamalla muutoksien kohteet.

4.2.2 Asennus- ja päivitysprosessi

Jokaista Anittan asennusta varten luotiin oma virtuaalinen Python-suoritusympäristö (engl. *Python virtual environment*). Virtuaaliympäristöllä eristettiin kaikki Anittan käyttämät Python-paketit muista Python-paketeista, joita saattoi tulla käyttöjärjestelmän mukana. Samalla varmistettiin, että kaikki Anitta asennukset olivat identtisiä suoritusympäristöstä riippumatta. Virtuaalisen suoritusympäristön luonnissa käytettiin env-moduulia, joka on Python versiosta 3.3 lähtien valmiiksi asennettu [42].

Kaikki Anittan ja sen eri moduulien tarvitsemat Python-paketit asennettiin Pythonin virallisesta PyPI (*Python Package Index*) paketti-indeksistä. Listausta pakettiriippuvuuksista päivitettiin ja ylläpidettiin *requirements.txt* tiedostossa, joka sijaitsi versionhallinnassa Anitta-projektin juuressa. Asennuksessa käytettiin *pip* Python-pakettienhallintatyökalua. Itse Anitta ja sen moduulit asennettiin eri tavalla. Ne ladattiin palvelimelle versiohallinnasta omaan hakemistoon, joka liitettiin virtuaaliympäristöön Pythonin polkutiedostolla [43]. Päivitysprosessi tapahtui lataamalla ensin uusimmat muutokset versionhallinnasta, jonka jälkeen Python-paketit päivitettiin *requirements.txt*:stä.

4.3 Ongelmat

Kehitysprosessin suurin ongelma oli testauksessa. Anittan koodipohjan kasvaessa, alkoi testaaminen manuaalisesti käydä työlääksi. Monimutkaisten ominaisuuksien kohdalla testaus vei paljon aikaa ja oli usein puutteellista. Manuaalinen regressiotestaus jokaisen muutoksen yhteydessä koettiin myös tylsäksi ja hankalaksi. Tämän seurauksena osa testitapauksista saatettiin ohittaa, sillä oletuksella että kyseiset testit eivät olleet oleellisia kyseiselle muutokselle. Testausta ei myöskään valvottu, saati pakotettu. Kovan kiireen alla se usein unohdettiin kokonaan tai jätettiin tarkoituksella tekemättä, jotta uusi versio Anittasta saataisiin nopeammin tuotantoon. Tämän seurauksena Anittan koodikanta oli ajoittain rikkinäinen ja pahimmillaan se päätyi tuotantoon asti.

Tuotantoon päässeiden koodivirheiden havaitseminen oli hidasta, koska tuotantoympäristöä ei aktiivisesti monitoroitu. Palvelimien terveydentilaa seurattiin manuaalisesti kehittäjien ja asiakkaan toimesta. Tämän seurauksena virheiden havaitsemiseen ja reagointiin saattoi kulua useita tunteja. Virheidenjäljitys (engl. *debugging*) oli työlästä, sillä virheisiin liittyviä virhelokeja ei ollut aina saatavilla tai ne olivat puutteellisia. Lokien etsimiseen kului usein myös paljon aikaa.

Kehitystiimin kasvaessa, kasvoi myös tuotantoon päässeiden koodivirheiden määrä, koska kehitysprosessin eri vaiheista ja niihin liittyvistä toimenpiteistä ei ollut tarkkaa dokumentaatiota. Tästä johtuen kehitysprosessi ei ollut yhtenäinen kehittäjien välillä. Uudet kehittäjät tekivät myös usein kehitystä eri tavalla kuin oli tarkoitus. Tämä johti usein virhetilanteisiin testauksessa ja uusien versioiden käyttöönotoissa.

5. PYTHON-KEHITYSYMPÄRISTÖN AUTOMATISOINTI

Anittan kehitysympäristön automatisointi aloitettiin jo muutama vuosi ennen diplomityön aloittamista vuoden 2017 syksyllä. Siitä lähtien kehitysympäristöä on kehitetty aktiivisesti vastaamaan paremmin kehittäjien tarpeita.

Päätavoite automatisoinnissa on yksitoikkoisten, virheherkkien ja puuduttavien rutiinitoimenpiteiden vähentäminen kehitysprosessista. Tällaisia toimenpiteitä ovat palvelimien tilan monitorointi, koodimuutoksien testaus ja uuden ohjelmaversion käyttöönotto. Kun rutiinitoimenpiteitä on vähemmän, jää kehittäjälle enemmän aikaa tuottavampaan tekemiseen kuten ohjelmointityöhön ja sen suunnitteluun. Muita tavoitteita ovat virheidenjäljityksen parantaminen ja kehitysprosessin nopeuttaminen.

Diplomityön tekemisen aikana kävi nopeasti ilmi, että aikaisemman kehityksen ratkaisut ovat samankaltaisia kuin jatkuvassa toimituksen käytännössä. Tämä luku käsitteleekin siis erästä toteutusta Python-kehitysympäristöstä, jossa hyödynnetään jatkuvaa toimitusta. Ratkaisut esitetään kronologisessa järjestyksessä vanhimmasta uusimpaan.

5.1 Työkalujen valintakriteerit

Työkaluja luvussa esiteltyjen ratkaisujen toteuttamiseen oli lähes rajattomasti, kuten myös Simo Mäkinen ja muut [1] ovat omassa tutkimustyössään huomanneet. Tässä työssä olennaista työkalujen valinnassa oli yhteensopivuus Python-kehitysympäristön kanssa, käyttöönoton helppous ja matalat käyttökustannukset.

Työkalujen tuli myös perustua avoimeen lähdekoodiin. Sen katsottiin takaavan työkalun kehitykselle paremman jatkuvuuden suljettuun lähdekoodiin verrattuna. Tietoturvan katsottiin lisäksi olevan parempaa avoimen lähdekoodin työkaluissa, koska lähdekoodit ovat kaikkien nähtävillä ja tarkistettavissa. Suljetun lähdekoodin työkaluissa lähdekoodien tarkistaminen ei ole mahdollista, mikä lisää epävarmuutta työkalun tietoturvaa kohtaan.

Pilvipalveluna tarjottavat työkalut rajattiin valintojen ulkopuolelle. Niiden katsottiin olevan käyttökustannuksiltaan korkeampia verrattuna paikallisesti asennettaviin ja käytettäviin työkaluihin. Lisäksi pilvipalveluihin liittyi epävarmuutta tietojen säilytyksen suhteen, sillä palveluissa ei ollut aina kerrottu tarkkaan missä ja miten palveluun annettuja tieto-

ja käsitellään sekä mihin ne tallennetaan. Tietoturvan kannalta tämä nähtiin uhkana, sillä tiedot voivat mahdollisesti vuotaa vääriin käsiin, mikäli palveluun murtaudutaan.

Omien työkalujen kehittämistä myös harkittiin, mutta siitä luovuttiin riittävän osaamisen puutteen vuoksi. Niiden ylläpito ja jatkokehitys olisi myös vaatinut enemmän henkilöresursseja kuin niitä olisi ollut käytettävissä.

5.2 Monitoroinnin kehittäminen

Ensimmäisenä kehitysympäristön automatisoinnissa otettiin käyttöön aktiivinen ohjelma- ja järjestelmätason monitorointi. Niiden puuttuminen kehitysympäristöstä katsottiin olevan kaikista vakavin ongelma, sillä kokonaan huomaamatta jääneet virheet Anittassa tai sitä ajavassa palvelimessa aiheuttivat eniten haittaa asiakkaille.

Ohjelmatason monitoroinnilla lyhennetään ohjelmavirheiden havaitsemiseen kuluvaa reagointiaikaa sekä parannetaan virhelokien kattavuutta ja keskitetään ne yhdestä paikasta saatavaksi.

Järjestelmätason monitoroinnilla varmistetaan, että palvelin, jolla Anittaa ajetaan, on aina käyttökunnossa. Tämä mahdollistaa myös ongelmatilanteiden ennaltaehkäisemisen kuten levytilan tai muistin loppumisen.

5.2.1 Järjestelmätason monitorointi

Järjestelmätason monitorointiin työkaluksi valittiin Zabbix 4.5, joka myöhemmin työn kirjoittamisen aikana päivitettiin versioon 5.0. Toisena vaihtoehtona monitorointiin oli Nagios.

Vertailtaessa molemmat työkalut olivat ominaisuuksiltaan samantasoisia ja yritystason järjestelmämonitorointiin suunniteltuja. Ratkaisevaksi tekijäksi valinnassa muodostui käyttöönoton helppous. Verrattuna Nagiokseen, Zabbixin asennusohjeet olivat selkeämmät ja yksinkertaisemmat. Nagioksen tarjoamat monet eri versiotyypit koettiin myös sekavaksi verrattuna Zabbixiin, joka tarjoaa vain yhden versiotyypin.

Käyttöönotto toteutettiin asentamalla Zabbixin ydin ja käyttöliittymä yhdelle palvelimelle. Zabbix-agentit asennettiin jokaiselle tuotanto- ja testipalvelimille. Asennuksissa noudatettiin Zabbixin virallisia asennusohjeita [44].

Zabbix-agentit konfigurointiin aktiiviseen toimintatilaan. Tämä soveltui parhaiten käyttötarkoitukseen, koska palvelimien palomuureihin ja verkkoympäristöön ei haluttu tehdä mitään muutoksia. Passiivisessa toimintatilassa niihin olisi pitänyt tehdä muutoksia, jotka sallivat Zabbix-palvelimen yhteydenotot agentteihin ulkoverkosta. Tämä nähtiin tietoturvan kannalta huonompana ratkaisuna.

Kuva 5.1: *Kuva toimiston fyysisestä monitorista.*

Aluksi Zabbixin havaitsemista ongelmista ilmoitettiin ylläpitäjille vain sähköpostitse. Tämä kuitenkaan ei aina riittänyt ongelmien huomaamiseen, sillä ilmoitukset hukkuivat helposti ylläpitäjien sähköpostilaatikkoon muiden viestien sekaan ja jäivät näin ollen korjaamatta.

Asian ratkaisemiseksi rakennettiin radiaattori (engl. *radiator*) eli niin kutsuttu fyysinen monitori, jonne palvelimissa havaitut ongelmat tulevat näkyviin reaaliajassa. Monitori sijoitettiin toimistossa näkyvälle paikalle, jotta ilmoitukset tulevat mahdollisimman nopeasti huomatuiksi. Radiaattorina (Kuva 5.1) toimii Raspberry Pi 3 ja siihen asennettu Raspberry Pi OS (entinen Rasbian) -käyttöjärjestelmä, joka selaimen kautta näyttää Zabbixin kojelautanäkymää (engl. *dashboard*).

5.2.2 Ohjelmatason monitorointi

Ohjelmatason monitorointiin työkaluksi valittiin Sentry 8.0. Valinnan tekeminen oli yksinkertaista, koska muita vastaavanlaisia avoimeen lähdekoodiin perustuvia työkaluja ei ollut päätöksentekohetkellä löydetty.

Sentry asennettiin yhdelle palvelimelle kontitettuun (engl. *containerization*) ajoympäristöön. Kontitetussa ajoympäristössä ohjelmat suoritetaan täysin eristettynä muista käyttöjärjestelmän ohjelmista, mikä tekee ohjelmien asentamisesta ja käyttämisestä helpompaa. Asennusta varten palvelin tarvitsi Docker-ohjelmiston. Sentryn sekä Dockerin asennuksessa noudatettiin ohjelmantarjoajien virallisia asennusohjeita [38, 45].

Jotta Anittassa tapahtuvat virheilmoitukset välittyvät Sentryyn, tarvitsi Anittan Python riippuvuuksiin lisätä *sentry-sdk* -paketti ja Djangon asetuksiin (settings.py tiedosto) lisätä alla esitetyt koodirivit (Ohjelma 5.1).

```
import os
import logging
import sentry_sdk
from . import __version__
from sentry_sdk.integrations.django import DjangoIntegration
from sentry_sdk.integrations.celery import CeleryIntegration
from sentry_sdk.integrations.logging import LoggingIntegration

sentry_logging = LoggingIntegration(
  level=logging.INFO,
  event_level=logging.WARNING
)

sentry_sdk.init(
  dsn=os.environ.get('DJANGO_RAVEN_DSN'),
  integrations=[sentry_logging, DjangoIntegration(), CeleryIntegration()],
  send_default_pii=True,
  release=__version__
)
```

Ohjelma 5.1: *Koodi Sentryn käyttöönottoa varten Djangossa.*

Tämän jälkeen Anittalle generoitiin DSN (*Data Source Name*) -osoite, jonka perusteella Anitta osaa lähettää virheilmoitukset Sentrylle. Generointi tehtiin ohjeiden mukaisesti Sentryn hallintapaneelista [46]. Palvelimilla DSN-osoite välitetään Anittalle ympäristömuuttujan kautta.

5.3 Testauksen automatisointi

Monitoroinnin kehityksen jälkeen keskityttiin testauksen automatisointiin. Testauksen automatisoinnilla varmistetaan, että testit suoritetaan aina jokaisen muutoksen yhteydessä ja että Anittan koodikanta on aina ehjässä tilassa. Lisäksi se estää tilanteet, jossa testaus unohdetaan tai tarkoituksella jätetään tekemättä.

Työkaluksi testauksen automatisointiin valittiin GitLab CI/CD. Sen avoimeen lähdekoodiin perustuva ilmainen versio oli ominaisuuksiltaan riittävän laaja työhön tarvittavien julkaisuputkien toteuttamiseen.

Suurimpana etuna muihin vastaaviin työkaluihin oli käyttöönoton helppous. Erillistä asennusta ei tarvinnut tehdä, sillä GitLab CI/CD löytyy sisäänrakennettu kaikista GitLab versioista. Myös laajat integraatiomahdollisuudet erilaisten viestintäpalvelujen kanssa kuten Slack, katsottiin eduksi.

Saatavilla oleva dokumentaatio oli myös kattava ja helposti ymmärrettävä, mikä helpotti julkaisuputkien rakentamista. Kehityksen jatkuvuuden kannalta GitLab CI/CD on myös sopiva valinta. Sitä kehitetään jatkuvasti ja on työn tekemisen aikana saanut paljon uusia ominaisuuksia. Tämä takaa, että julkaisuputkia voidaan tulevaisuudessa kehittää vieläkin paremmaksi.

Toisena vaihtoehtona työkaluksi testauksen automatisointiin pohdittiin Jenkinsiä. Sen käyttöönotto olisi kuitenkin vaatinut erillisen asennuksen toisin kuin GitLab CI/CD. Käyttäminen ei ollut myöskään yhtä yksinkertaista kuin GitLab CI/CD:ssä ja sen opetteleminen olisi todennäköisesti vienyt enemmän aikaa.

5.3.1 GitLab CI/CD asentaminen

GitLab CI/CD asennusta varten asennettiin kaksi GitLab Runner -suorittajaa. Suorittajat asennettiin hajautetusti kahdelle palvelimelle, yksi yrityksen toimistoon ja toinen pilvipalveluun. Hajautuksella varmistettiin, että GitLab CI/CD on aina käytettävissä, mikäli jompikumpi suorittajista ei vastaa esimerkiksi yhteyden ollessa poikki.

Asennukset tehtiin kontitettuun ajoympäristöön. Tämän katsottiin olevan helpointa muihin asennusmuotoihin verrattuna. Sitä varten palvelimille asennettiin ensin Docker-ohjelmisto. Suorittajien sekä Dockerin asennuksessa noudatettiin ohjelmantarjoajien virallisia asennusohjeita. [47, 45]

GitLab instanssin URL	https://git.onebyte.fi
GitLab CI tokeni	xxxxxxxxxxxxxx
Suorittajan selite	runner0.git.onebyte.fi
Suorittajan tagit	upcloud, docker
Suorittajantyyppi	docker

Taulukko 5.1: *Suorittajan runner0.git.onebyte.fi rekisteröintiarvot.*

GitLab instanssin URL	https://git.onebyte.fi
GitLab CI tokeni	xxxxxxxxxxxxxx
Suorittajan selite	toimistorunner0.onebyte.toimisto
Suorittajan tagit	tampere, docker
Suorittajantyyppi	docker

Taulukko 5.2: *Suorittajan toimistorunner0.onebyte.toimisto rekisteröintiarvot.*

Asennuksen yhteydessä suorittajat rekisteröitiin GitLabiin. Tämä vaadittiin, jotta suorittajat voivat vastaanottaa ja suorittaa niille annettuja tehtäviä. Taulukoissa 5.1 ja 5.2 on esitetty rekisteröinnissä käytetyt arvot. Rekisteröintiä varten vaadittiin GitLab CI tokeni, joka löytyi GitLabin ylläpitäjien hallintapaneelin kautta. Kun suorittajat oli rekisteröity, varmistettiin vielä hallintapaneelista, että ne olivat rekisteröityneet oikein.

5.3.2 Automaatiotestit

Testauksen automatisointia varten Anittaan piti kirjoittaa automaatiotestejä. Aluksi testit kirjoitettiin Pythonin unittest-kirjastolla, koska se oli Djangon virallisesti ohjeiden mukainen suositus [48]. Myöhemmin testien kirjoittamisessa käytettiin myös pytest-kirjastoa. Sen todettiin soveltuvan paremmin joidenkin testien kirjoittamiseen.

Julkaisuputkessa testit suoritetaan *tox* Python-työkalulla. Se tarjoaa yksinkertaisen rajapinnan testien suorittamiseen [49]. Testit ajetaan yhdellä tox-komennolla. Tämä vähentää huomattavasti testaukseen tarvittavien komentojen määrää tehden julkaisuputken testausvaiheesta yksinkertaisen.

5.4 Käyttöönoton automatisointi

Testauksen automatisoinnin jälkeen automatisoitiin käyttöönotto. Kuten manuaalinen testaus, oli myös manuaalinen käyttöönotto hidasta ja virheherkkää. Automatisoinnin avulla varmistetaan, että käyttöönotto tulee joka muutoksen kohdalla tehtyä aina samalla tavalla. Tämä poistaa mahdollisuuden inhimillisien virheiden esiintymisen käyttöönotossa ja tekee siitä nopeampaa.

Käyttöönoton automatisointi toteutettiin työntö-tyyppisesti (engl. *push*) lisäämällä julkaisuputkeen käyttöönottovaihe. Tässä vaiheessa luodaan yhteys palvelimeen, mikä laukaisee palvelimella päivitysprosessin. Yhteys muodostetaan SSH-protokollalla ja tunnistautumiseen käytetään julkisen avaimen infrastruktuuriin (engl. *PKI* tai *Public Key Infrastructure*) kuuluvaa SSH-avainta. Avaimen julkinen osa (engl. *public key*) liitettiin palvelimella anitta-käyttäjän *authorized_keys* tiedostoon rivi seuraavanlaisesti:

```
command="~/paivita.sh",no-port-forwarding,no-x11-forwarding,no-pty [JULKINEN AVAIN]
```

Kun yhteys muodostetaan julkista avainta vastaavalla yksityisellä avaimella (engl. *private key*), suorittaa palvelin aina komennon *~/paivita.sh*. Yksityinen avain välitetään julkaisuputkelle ympäristömuuttujien avulla, joita hallitaan GitLabin projektiasetuksista [50]. Muita, kuin edellä mainittua komentoa, ei avaimella voi suorittaa. Tietoturvan kannalta tämä on järkevä ratkaisu, sillä jos yksityinen avain vuotaa pahantahtoisille tahoille, eivät he pysty kuin päivittämään palvelimen.

5.5 Asennus- ja päivitysprosessin parantaminen

Käyttöönoton automatisoinnin jälkeen, parannettiin asennus- ja päivitysprosessia. Tätä varten Anittasta ja sen moduuleista luotiin Python-paketit. Paketoinnilla yksinkertaistettiin ja nopeutettiin molempia prosesseja. Lisäksi luotiin erillinen Anitta kokoonpanopaketti, jolla yhdistettiin Anitta, sen moduulit ja näiden pakettiriippuvuudet yhden paketin kautta helposti asennettavaksi ja ylläpidettäväksi.

Python pakettien tallennusta varten tarvittiin erillinen yksityinen Python paketti-indeksi. Työkaluksi valittiin PyPICloud. Sen käyttöönotto oli yksinkertaista ja tarjosi tarpeeksi laajan käyttöoikeuksienhallinnan (engl. *Access Control Management*), jotta eri asiakkaiden kokoonpanopaketit saatiin eristettyä toisistaan. Toisena vaihtoehtona vastaavanlaisena vaihtoehtona oli Devpi, mutta sen käyttöönotto todettiin liian hankalaksi ja käyttöoikeuksienhallinta oli epäselvä.

5.5.1 Anittan ja sen moduulien paketointi

Anittan ja sen moduulien paketointia varten määritettiin *setup.py*-tiedosto, jonka perusteella Python-paketit luodaan. Tiedosto lisättiin versionhallintaan jokaiseen projektin juureen alla olevan esimerkin (Ohjelma 5.3) mukaisesti.

```
import os
import versioneer
from setuptools import setup, find_packages

# Poimi muut kuin tyhjat ja kommenttirivit
requirements = [
  rivi.strip()
  for rivi in open(
    os.path.join(os.path.dirname(__file__), 'requirements.txt')
  )
  if not rivi.startswith('#')
  and rivi.strip()
]

setup(
  name='[PAKETIN NIMI]',
  version=versioneer.get_version(),
  cmdclass=versioneer.get_cmdclass(),
  description='[PAKETIN KUVAUS]',
  url='[URL-osoite]',
  author='OneByte Oy',
  author_email='yllapito@onebyte.fi',
  packages=find_packages(),
  include_package_data=True,
  install_requires=[rivi for rivi in requirements],
  zip_safe=False
)
```

Ohjelma 5.2: *Esimerkki setup.py sisällöstä Python-pakettien luontia varten.*

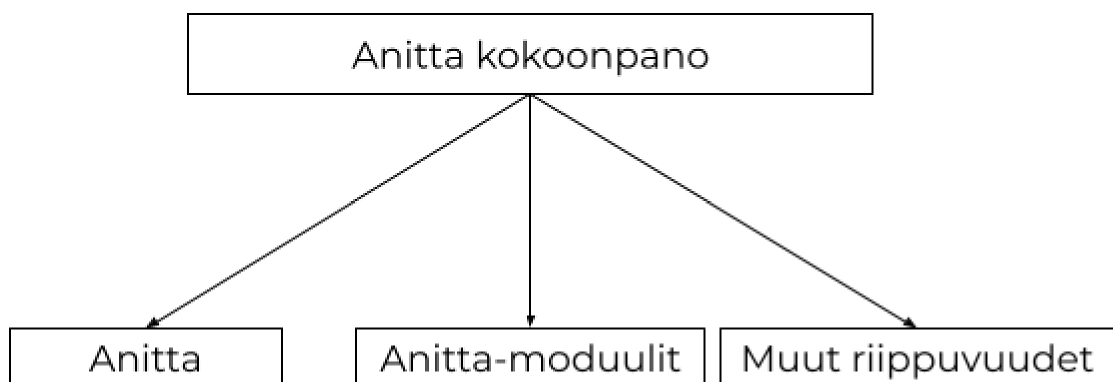
Riveillä 5 - 12 käsitellään Python-paketin pakettiriippuvuudet, jotka määritetään *requirements.txt* tiedostossa projektin juuressa. Riveillä 15 - 16 määritetään paketin nimi ja versionumero. Riveillä 19 - 20 määritetään lyhyt kuvaus paketista ja URL-osoite projektiin versionhallinnassa.

Paketointiin ja julkaisuun käytettiin *twine* Python-työkalua. Paketin versionumeron dynaamiseen määrittämiseen käytettiin *versioneer* Python-työkalua. Versioneerissa versionumero määritetään versionhallinnan tagin perusteella. Tämä helpottaa huomattavasti uu-

den version julkaisua, sillä siihen riittää vain tagin luonti versionhallintaan. Versioneerin asennus tehtiin jokaiseen projektiin työkalun virallisten ohjeiden mukaan [51]. Versiointityyliksi valittiin semanttinen versiointityyli [52].

5.5.2 Kokoonpanopaketti

Anittan, sen moduulien ja näiden pakettiriippuvuuksien ylläpidon sekä asentamisen helpottamiseksi luotiin kokoonpanopaketti. Paketti on tavallinen Python-paketti, joka sisältää pakettiriippuvuuksina tiedon edellä mainituista Python-paketeista, joita tarvitaan toimivan Anitta-kokoonpanon asentamiseen. Tämä on havainnollistettu kuvassa 5.2.



Kuva 5.2: *Kuvaus kokoonpanopaketin riippuvuuksista.*

Kokoonpanopaketti toteutettiin jokaiselle eri Anittan kokoonpanolle luomalla versionhallintaan oma projekti. Projekti sisältää edellisessä aliluvussa kerrotut *setup.py* ja *requirements.txt* tiedostot paketin luontia varten. Aina kun Anittaa päivitetään, se tehdään kokoonpanoprojektin kautta pakettiriippuvuuksia muuttamalla.

5.5.3 PyPICloudin asentaminen

PyPICloud asennus tarvitsi yhden palvelimen. Asennus suoritettiin virallisten ohjeiden mukaisesti [53]. Pakettien tallennusvaihtoehdoksi valittiin paikallinen levy. Toisena vaihtoehtona oli pilvitallennuspalveluiden kuten Google Cloud Storage tai Amazon S3 käyttäminen, mutta niiden käytöstä ei ollut tarpeeksi tietämystä valintaa tehdessä.

Asennuksen jälkeen luotiin ryhmät ja käyttäjätunnukset julkaisuputkia varten. Käyttäjätunnukset välitetään julkaisuputkille ympäristömuuttujien kautta.

5.6 Julkaisuputkien esittely

Työtä varten rakennettiin kaksi julkaisuputkea. Alla olevassa taulukossa (Taulukko 5.3) on esitetty putkista lyhyet kuvaukset, määritystiedostojen nimet.

Nimi	Määritystiedosto	Lyhyt kuvaus
Pakettiputki	pipeline_python_release.yml	Testaa, paketoi ja julkaisee Anittan ja sen moduulit.
Kokoonpanoputki	pipeline_anitta_kokoonpano.yml	Testaa, paketoi, julkaisee ja käyttöönottaa Anitta kokoonpanot.

Taulukko 5.3: *Julkaisuputket, joita käytetään Anittan kehitysprosessissa.*

Putkien määritystiedostot tallennettiin versionhallintaan *ci-mallit* nimiseen projektiin. Putket käyttöönotettiin luomalla jokaisen kohdeprojektin juureen *.gitlab-ci.yml*-tiedosto, jossa käytettävä putki sisällytettiin projektiin GitLab CI/CD:n *include*-direktiivillä seuraavanlaisesti:

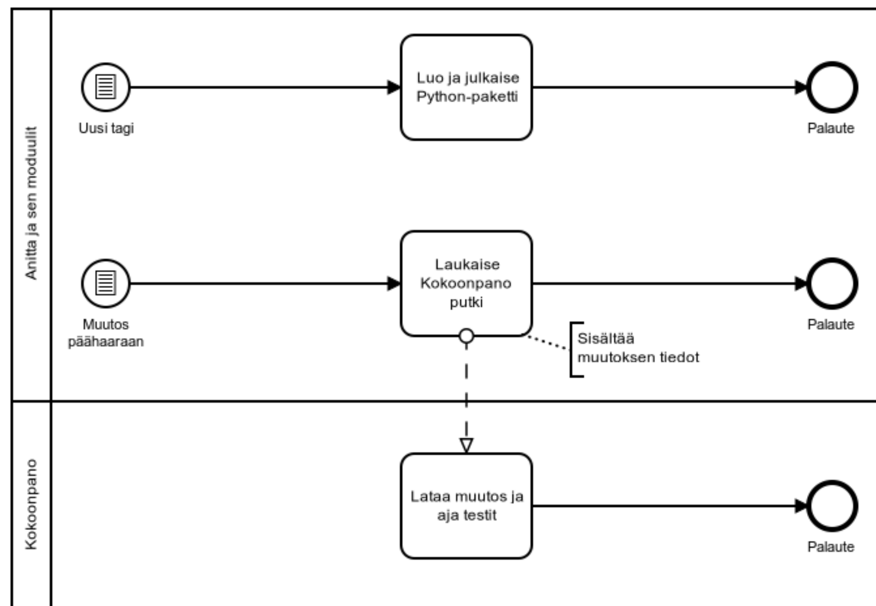
```
include:
  - project: 'onebyte/ci-mallit'
    file: '/pipeline_anitta_kokoonpano.yml'
```

Ohjelma 5.3: *Esimerkki .gitlab-ci.yml jossa putki sisällytetään projektiin.*

Tämän todettiin olevan ylläpidon kannalta paras ratkaisu. Näin putkiin tehtäviä muutoksia ei tarvitse toistaa jokaisen kohdeprojektin kohdalla erikseen vaan ne voidaan keskitetysti toteuttaa *ci-mallit* -projektin kautta. Putkien sisäistä toimintaa ohjataan tarvittaessa ympäristömuuttujien kautta, jotka määritetään jokaiselle projektille erikseen GitLabin projektiasetuksista.

Kehittäjien tarpeesta riippuen, GitLab ilmoittaa putkien palautteen joko Slackiin tai sähköpostiin. Määritykset tehtiin jokaiselle projektille erikseen GitLabin projektiasetuksista.

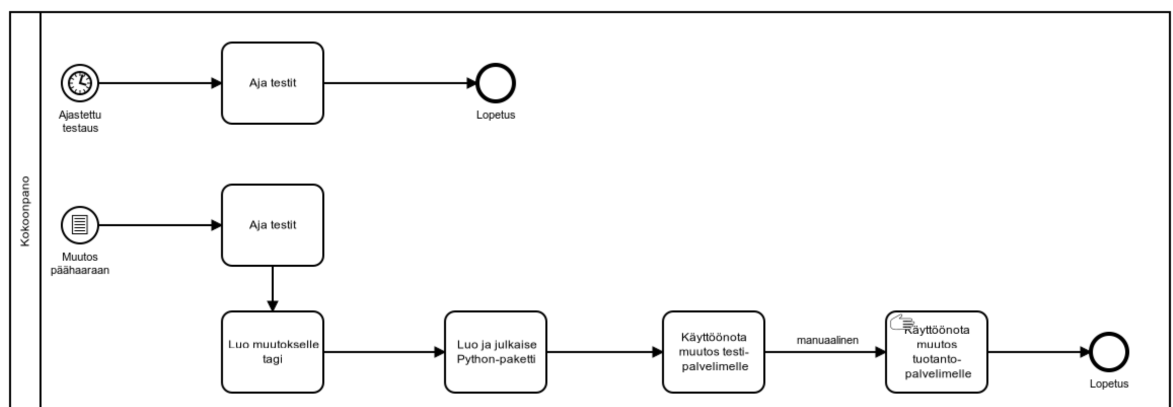
Pakettiputki koostuu kahdesta eri suorituspolusta. Suorituspolut on kuvattu alla olevassa BPMN kuvaajassa (Kuva 5.3).



Kuva 5.3: *BPMN kuvaaja Pakettiputken toiminnasta.*

Pakettiputken ensimmäinen suorituspolku alkaa muutoksen työntämisestä projektin päähaaraan. Tällöin muutos testataan uusimmassa Anitta-kokoonpanossa. Tämä tapahtuu laukaisemalla Anitta-kokoonpanon putki ja välittämällä parametreina testattavan muutoksen tiedot. Toinen suorituspolku alkaa, kun projektiin luodaan uusi tagi. Tällöin kyseisestä tagin sisältävästä muutoksesta luodaan Python-paketti ja se julkaistaan PyPICloudiin.

Kokoonpanoputki koostuu kahdesta eri suorituspolusta. Suorituspolut on kuvattu alla olevassa BPMN kuvaajassa (Kuva 5.4).



Kuva 5.4: *BPMN kuvaaja Kokoonpanoputken toiminnasta.*

Kokoonpanoputken ensimmäinen suorituspolku alkaa ajastetusti ja suorittaa tällöin Anitta kokoonpanon testauksen. Toinen suorituspolku alkaa, kun kokoonpanoon työnnetään muutos. Tällöin muutos ensin testataan, jonka jälkeen siihen lisätään tagi. Seuraavaksi

luodaan ja julkaistaan Python-paketti, joka käyttöönotetaan automaattisesti testipalvelimella. Kun kehittäjät ovat testanneet muutoksen toimivuuttaa testipalvelimella, se otetaan käyttöön manuaalisella hyväksynnällä tuotantopalvelimella.

6. ARVIOINTI JA JATKOKEHITYS

Päätavoitteena työssä oli kehitysympäristön automatisoinnin lisääminen, mikä tarkoitti yksitoikkoisten, virheherkkien ja puuduttavien rutiinitoimenpiteiden vähentämistä kehitysprosessissa. Näihin toimenpiteisiin lukeutui palvelimien tilan monitorointi, koodimuutoksien testaus ja uuden ohjelmaversion käyttöönotto. Muita tavoitteita työssä oli virheidenjäljityksen ja kehitysprosessin nopeuden parantaminen.

Työ onnistui kaikissa sille asetetuissa tavoitteissa. Niiden täyttäminen oli helppoa, sillä jatkuvan ohjelmistosuunnittelun käytäntöjä on viime vuosikymmeninä tutkittu hyvinkin laajasti ja aiheeseen liittyvää aineistoa oli paljon saatavilla. Lisäksi työkaluja työn toteuttamiseen oli runsaasti tarjolla. [1, 3]

Työn varsinaiseen toteutukseen kului yhteensä noin 2,5 vuotta. Suurin osa ajasta kului julkaisuputkien rakentamiseen. Tämä oli myös työssä kaikista haastavin osuus. GitLab CI/CD ei tarjonnut työn tekemisen aikana riittäviä testaustyökaluja julkaisuputkien kehittämiseen. Julkaisuputket otettiin käyttöön ilman testausta, minkä takia etenkin työn alkuvaiheilla putket olivat usein rikki niihin liittyvien ohjelmointivirheiden takia.

6.1 Lopputulos

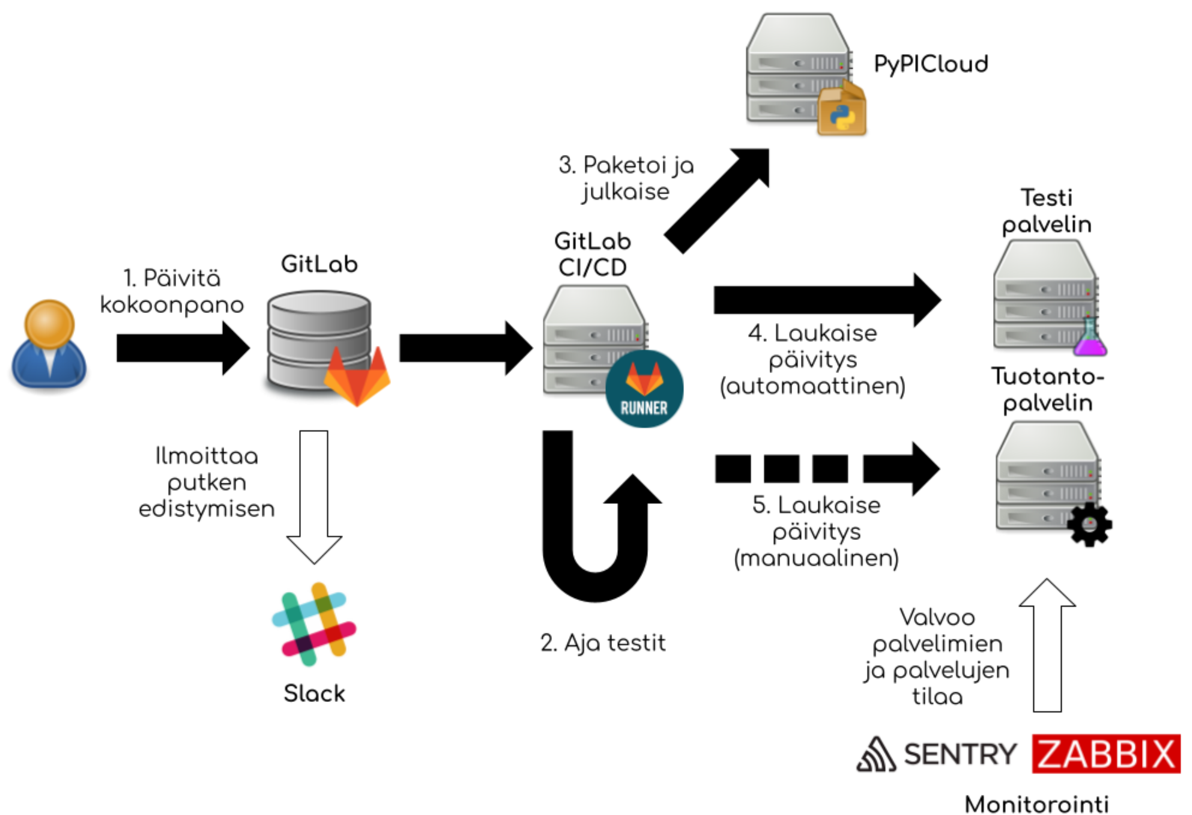
Nykyisin Anittan koodikanta on hyvin laaja-alainen ja sen kehityksestä vastaa yli viisihenkinen kehitystiimi. Tiimissä on kaksi ylläpitäjää, johon diplomityön kirjoittaja kuuluu, ja loput ovat kehittäjiä.

Kehitysympäristö on aikaisempaa ympäristöä laajempi ja sisältää monitoroinnin sekä testauksen ja käyttöönoton automatisointiin tarvitut palvelimet aikaisempien lisäksi. Lähdekoodien tietovarastona toimii edelleen GitLab-versionhallintajärjestelmän Community Edition -versio.

Kehitysprosessissa koodimuutoksien hallintaan käytetään vielä haarautusmallia. Vaikka jatkuvan integroinnin käytäntö suosittele sen käyttämistä, ei käytön lopettamiselle nähty tarpeeksi painavia perusteita. Lisäksi lopettaminen olisi aiheuttanut muutosvastarintaa kehittäjien keskuudessa.

Nykyinen kehitysprosessi on jatkuvan toimituksen kaltainen. Jokainen Anittaan ja sen moduuleihin tehtävät muutokset testataan automaattisesti aina uusimmassa Anitta-kokoonpanossa. Kun testit on suoritettu onnistuneesti, kehittäjä luo manuaalisesti muutokselle tagin, jonka jälkeen se automaattisesti paketoidaan ja julkaistaan paketti-indeksiin.

Anittan ja sen moduuleista julkaistut uudet versiot päivitetään tämän jälkeen Anittan kokoonpanoon. Kokoonpanoon tehty muutos testataan automaattisesti. Onnistuneen testauksen jälkeen muutokselle luodaan automaattisesti tagi, jonka jälkeen se paketoidaan ja julkaistaan paketti-indeksiin. Sen jälkeen julkaistu kokoonpanopaketti päivitetään testipalvelimelle kehittäjien testattavaksi. Kun kehittäjät ovat testanneet muutoksen toimivuuden testipalvelimella, hyväksyvät he manuaalisesti kokoonpanon päivityksen tuotantopalvelimelle. Kuvassa 6.1 on esitetty nykyinen kehitysprosessi.



Kuva 6.1: *Kuvaus Anittan nykyisestä kehitysprosessista.*

Suurimpana muutoksena aikaisempaan kehitysprosessiin (Kuva 4.2) on testauksen ja käyttöönoton vastuun siirtyminen pois kehittäjiltä. Kehittäjien ei tarvitse enää muistaa tai edes tietää miten testaus tai käyttöönotto tehdään ja voivat näin ollen keskittyä enemmän itse ohjelmointiin. Ylläpidollisesti Anittan asentaminen ja päivittäminen on paljon helpompaa ja nopeampaa, sillä ne tehdään kokoonpanopaketilla, mikä vähentää virheiden mahdollisuuksia käyttöönotoissa.

Parannukset monitoroinnissa nopeuttavat virheiden havaitsemista Anittassa ja sitä ajavien palvelimien toiminnassa. Ilmoitukset virheistä ja niihin liittyvistä lokitiedoista näkyvät keskitetysti yhdestä paikkaa, mikä helpottaa virheidenjäljittämistä ja korjaamista.

6.2 Tulokset

Kokoonpanoputken ja Pakettiputken eri vaiheisiin kuluvat suoritusajat ovat listattuna alla olevissa taulukoissa (Taulukko 6.1 ja Taulukko 6.2). Tulokset kerättiin GitLabin käyttöliittymän analytiikkasivuilta.

Vaihe	Suoritusaika
Testaus	noin 11 minuuttia
Paketointi ja julkaisu	30 sekuntta
Käyttöönotto testi- ja tuotantopalvelimille	noin 0,5 - 5 minuuttia

Taulukko 6.1: *Kokoonpanoputken vaiheiden suoritusajat.*

Vaihe	Suoritusaika
Testaus	noin 11 minuuttia
Paketointi ja julkaisu	30 sekuntia

Taulukko 6.2: *Pakettiputken vaiheiden suoritusajat.*

Testauksen osalta molempien putkien suoritusajat ovat samat, koska kaikki testaukset suoritetaan kokoonpanojen kautta. Myös paketoinnin ja julkaisun suoritusajat ovat samat, koska kyseessä on sama tehtävä. Kokoonpanoputkessa käyttöönotto testi- ja tuotantopalvelimille vaihteli päivityksen koosta riippuen. Nopeimmillaan se kesti 30 sekuntia ja hitaimmillaan noin 5 minuuttia.

Kahden vuoden aikana OneByten oman Anitta-kokoonpanon Kokoonpanoputki keskeytyi 227 kertaa 1585 kerrasta. Putken onnistumisprosentti on 83,48 %. Etenkin työn alussa keskeytyksien syynä oli usein itse putken kehitykseen liittyviä ohjelmointivirheitä. Karkeasti vain noin 30 tapauksessa syynä oli testauksen epäonnistuminen. Näistä tilanteista vain noin puolessa oli kyseessä todellinen ohjelmointivirhe, joka olisi aiheuttanut tuotannon kaatumisen. Tulokset eivät ole vertailukelpoisia aikaan ennen työn aloittamista, koska kyseiseltä ajalta ei ole tilastoitua tietoa onnistuneista tai epäonnistuneista päivityksistä Anittaan.

Kehittäjien subjektiivinen kokemus julkaisuputkien toiminnasta oli pääosin myönteinen. Erityisesti manuaalisen työn vähentyminen testauksen, käyttöönoton ja monitoroinnin osalta koettiin positiivisena parannuksena aikaisempaan kehitysprosessiin.

6.3 Jatkokehitys

Kuten kaikessa, tässäkin työssä on omat heikkoutensa. Tuloksien perusteella suurimpina heikkouksina todettiin testauksen pitkät suoritusajat sekä yksipuolisuus. Tällä hetkellä testauksen suoritusaika on noin 11 minuuttia. Jatkuvan integroinnin periaatteiden mukaisesti sen tulisi olla alle 10 minuuttia.

Testaus on myös yksipuolista, sillä Anitta-moduuleille ei ole ollenkaan yksikkötestejä, vaan testaus suoritetaan aina uusimmissa Anitta-kokoonpanossa integraatiotestauksen muodossa. Tämä on osaltaan syynä testauksen pitkiin suoritusaikoihin, koska integraatiotestaus on hitaampaa kuin yksikkötestaus.

Testausta tulisi kehittää niin virheet havaittaisiin aikaisemmassa vaiheessa kehitystä. Aina kun integraatiotesteissä huomataan virhe, tulisi mahdollisuuksien mukaan luoda yksikkötestejä, jotka huomaavat kyseisen virheen. Näin toimimalla saadaan testauksen rakennetta muutettua testauspyramidin mukaiseksi ja samalla testauksen suoritusaikoja pienemmiksi.

Myöhemmin julkaisuputkiin on myös tarkoitus lisätä koodikattavuuden ja koodityylin testaus. Ne auttavat osaltaan mahdollisten ohjelmavirheiden havaitsemisessa ja korjaamisessa.

Anittan kehityksessä on periaatteessa mahdollista soveltaa jatkuvaa käyttöönottoa, jossa jokainen muutos käyttöönotettaisiin automaattisesti tuotantoon ilman manuaalista hyväksyntään. Tämä kuitenkin vaatisi, että Anittan testaus olisi nykyistä kattavampaa, jotta virheellisiä muutoksia pääsisi tuotantoon mahdollisimman vähän.

Työssä toteutettua kehitysympäristöä voidaan jo nyt käyttää myös yrityksen muihin ohjelmistoprojekteihin. Tarkoitus olisi, että tulevaisuudessa uudet Python-projektit hyödyntäisivät sitä kehityksessä.

7. YHTEENVETO

Työssä automatisoitiin aiemmin perinteiseen ohjelmistokehitykseen perustuvaa Python-kehitysympäristöä vastamaan paremmin nykyaikaista ohjelmistokehitystä. Samalla parannettiin kehitysprosessin virheidenjäljitystä ja nopeutta. Kehitysympäristön kehittäminen alkoi muutama vuosi ennen diplomityön aloittamista ja se sovelsi jatkuvan toimituksen ohjelmistosuunnittelu käytäntöä.

Työssä kehitystä jatkettiin eteenpäin, jonka lopputuloksena kehitysympäristöä saatiin automatisoitua kahden julkaisuputken avulla ja kehitysprosessin virheidenjäljitystä parannettua monitoroinnilla. Näiden yhteistuloksena myös kehitysprosessin nopeus parantui. Alla olevassa taulukossa (Taulukko 7.1) on esitelty työn kannalta tärkeimmät tiedot

Aihe	Python-kehitysympäristön automatisointi
Kohdejärjestelmä	Anitta (Laskutus- ja perintäjärjestelmä)
Päätyökalu	GitLab CI/CD
Monitorointityökalut	Zabbix ja Sentry
Muut työkalut	PyPICloud
Julkaisuputken suoritusaika	noin 12 - 16,5 minuuttia
Julkaisuputken testausvaiheen suoritusaika	noin 11 minuuttia
Vahvuudet	Automaattinen testaus ja käyttöönotto sekä aktiivinen monitorointi
Heikkoudet	Testien yksipuolisuus ja testauksen hitaus

Taulukko 7.1: *Yhteenvetotaulukko.*

Julkaisuputkien ansiosta testaus- ja käyttöönottovaihetta ei tarvitse suorittaa manuaalisesti. Tämä vähensi kyseisissä vaiheissa tapahtuneiden virheiden määrää. Samalla vastuu vaiheiden suorittamisesta siirtyi pois kehittäjiltä, jolloin he pystyivät aiempaa paremmin keskittymään itse ohjelmointiin.

Monitoroinnin ansiosta virheidenjäljitys helpottui huomattavasti. Kehittäjät näkevät keskitetysti yhdestä paikkaa kaikki ohjelmavirheisiin liittyvät lokitiedot, eikä niitä tarvitse enää hakea jokaiselta palvelimelta erikseen. Myös ylläpitäjät voivat seurata palvelimien resurssien käyttöä keskitetysti ja saavat ilmoituksen sähköpostiin sekä toimistolla sijaitsevaan radiaattoriin, kun palvelimien tilassa havaitaan ongelmia.

Saatujen tuloksien perusteella kehitysympäristö ei vastannut täysin jatkuvalle toimitukselle asetettuja määrityksiä. Testauksen osalta ongelmaksi osoittautui testien yksipuolisuus

sekä liian pitkät suoritusajat. Lisäksi suositusten vastaisesti koodimuutoksien hallintaan käytettiin edelleen haarautusmallia.

Jatkokehityksen kannalta on tärkeää, että testauksen suoritusajat saadaan lyhyiksi. Yksikkötestausta tulisi hyödyntää nykyistä enemmän, jotta ohjelmavirheet huomattaisiin aikaisemmassa vaiheessa kehitystä. Tämä muuttaisi testauksen rakennetta enemmän testauspyramidin mukaiseksi ja pienentäisi samalla testauksen suoritusaikoja.

Työssä toteutettua kehitysympäristöä voidaan käyttää myös muihin ohjelmistoprojekteihin. Tulevaisuudessa yrityksen uusien Python-projektien olisikin tarkoitus hyödyntää sitä kehityksessä.

LÄHTEET

[1] S. Mäkinen, M. Leppänen, T. Kilamo, A.-L. Mattila, E. Laukkanen, M. Pagels, and T. Männistö, "Improving the delivery cycle: A multiple-case study of the toolchains in finnish software intensive enterprises," *Information and Software Technology*, vol. 80, pp. 175 – 194, 2016. Saatavissa: http://www.sciencedirect.com/science/article/pii/S0950584916301434

[2] Jez Humble, "What is Continuous Delivery? - Continuous Delivery," 2017, [Viitattu 28.1.2020]. Saatavissa: https://continuousdelivery.com/

[3] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[4] M. Shahin, M. Zahedi, M. A. Babar, and L. Zhu, "An empirical study of architecting for continuous delivery and deployment," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1061–1108, 2019.

[5] M. Leppänen, S. Mäkinen, M. Pagels, V. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," *IEEE Software*, vol. 32, no. 2, pp. 64–72, 2015.

[6] J. Humble and D. Farley, "Continuous delivery," p. 463 s., 2011, lisäpainokset: 7th pr. 2013.; Kannessa alanimeke: Reliable software release through build, test, and deployment automation.

[7] M. Fowler, "DeploymentPipeline," 2013, [Viitattu 16.1.2020]. Saatavissa: https://martinfowler.com/bliki/DeploymentPipeline.html

[8] Jez Humble, "Continuous Testing - Continuous Delivery," 2017, [Viitattu 6.1.2021]. Saatavissa: https://continuousdelivery.com/foundations/test-automation/

[9] H. Juha, "Regressiotestaus ja testien valintatekniikat," 2005, [Viitattu 22.4.2021]. Saatavissa: http://www.cs.uku.fi/tutkimus/sose/material/Regressiotestaus.pdf

[10] M. Fowler, "TestPyramid," 2012, [Viitattu 12.11.2020]. Saatavissa: https://martinfowler.com/bliki/TestPyramid.html

[11] James Crisp, "Automated Testing and the Test Pyramid – James Crisp," 2011, [Viitattu 7.1.2021]. Saatavissa: https://jamescrisp.org/2011/05/30/automated-testing-and-the-test-pyramid/

[12] Puppet Labs, "2014 State of DevOps Report," 2014, [Viitattu 1.7.2019]. Saatavissa: https://puppet.com/resources/whitepaper/2014-state-devops-report/thank-you

[13] M. Fowler, "ContinuousIntegrationCertification," 2017, [Viitattu 10.11.2020]. Saatavissa: https://martinfowler.com/bliki/ContinuousIntegrationCertification.html

[14] Jez Humble, "Continuous Integration - Continuous Delivery," 2017, [Viitattu 14.1.2020]. Saatavissa: https://continuousdelivery.com/foundations/continuous-integration/

[15] M. Fowler, "FeatureBranch," 2009, [Viitattu 24.11.2020]. Saatavissa: https://martinfowler.com/bliki/FeatureBranch.html

[16] GitLab Inc., "CI/CD pipelines | GitLab," [Viitattu 12.1.2021]. Saatavissa: https://docs.gitlab.com/ee/ci/pipelines/

[17] Wikipedia, "Continuos delivery," 2019, [Viitattu 13.7.2019]. Saatavissa: https://en.wikipedia.org/wiki/Continuos_delivery

[18] M. B. ThoughtWorks, "5 Traits of a Good Delivery Pipeline," 2016, [Viitattu 29.1.2020]. Saatavissa: https://www.thoughtworks.com/insights/blog/5-traits-good-delivery-pipeline

[19] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, Mar 2015.

[20] M. Fowler, "ContinuousDelivery," 2013, [Viitattu 16.1.2020]. Saatavissa: https://martinfowler.com/bliki/ContinuousDelivery.html

[21] Red Hat, "What is CI/CD?" [Viitattu 30.1.2020]. Saatavissa: https://www.redhat.com/en/topics/devops/what-is-ci-cd

[22] Jez Humble, "Continuous Delivery vs Continuous Deployment? - Continuous Delivery," 2017, [Viitattu 30.1.2020]. Saatavissa: https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/

[23] S. Labs, "Choosing a CI-CD Tool | Sauce Labs," 2018, [Viitattu 31.5.2021]. Saatavissa: https://saucelabs.com/blog/choosing-a-ci-cd-tool

[24] Badri N Srinivasan , "What Tools Do You Need for Continuous Delivery? - DZone DevOps," [Viitattu 9.4.2021]. Saatavissa: https://dzone.com/articles/101-tools-for-continuous-delivery

[25] GitLab Inc., "Introduction to CI/CD with GitLab | GitLab," [Viitattu 26.2.2020]. Saatavissa: https://docs.gitlab.com/12.7/ee/ci/introduction/index.html

[26] GitLab Inc., "History of GitLab | GitLab," [Viitattu 15.2.2020]. Saatavissa: https://about.gitlab.com/company/history/

[27] GitLab Inc., "GitLab Analyst Report | GitLab," [Viitattu 13.3.2020]. Saatavissa: https://about.gitlab.com/analysts/forrester-ci/

[28] GitLab Inc., "GitLab CI/CD | GitLab," [Viitattu 25.2.2020]. Saatavissa: https://docs.gitlab.com/12.7/ee/ci/README.html

[29] GitLab Inc., "GitLab Continuous Integration and Delivery | GitLab," [Viitattu 11.5.2019]. Saatavissa: https://about.gitlab.com/product/continuous-integration/

[30] GitLab Inc., "GitLab Runner Docs | GitLab," [Viitattu 15.2.2020]. Saatavissa: https://docs.gitlab.com/runner/

[31] GitLab Inc., "Local runner execution MVC (2797) · Issues · GitLab.org / gitlab-runner · GitLab," [Viitattu 25.2.2020]. Saatavissa: https://gitlab.com/gitlab-org/gitlab-runner/issues/2797

[32] Martin Heller, InfoWorld, "What is Jenkins? The CI server explained | InfoWorld," [Viitattu 7.5.2021]. Saatavissa: https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html

[33] Dmitry Lambert, Zabbix LLC, "How to deploy Zabbix on PostgreSQL with Timescale DB plugin – Zabbix Blog," [Viitattu 7.5.2021]. Saatavissa: https://blog.zabbix.com/how-to-deploy-zabbix-on-postgresql-with-timescale-db-plugin/13668/

[34] Zabbix LLC, "Zabbix Screenshots," [Viitattu 7.5.2021]. Saatavissa: https://www.zabbix.com/en/screenshots

[35] Dmitry Lambert, Zabbix LLC, "Zabbix Agent: Active vs. Passive – Zabbix Blog," [Viitattu 7.5.2021]. Saatavissa: https://blog.zabbix.com/zabbix-agent-active-vs-passive/9207/

[36] Zabbix LLC, "Zabbix Agent," [Viitattu 7.5.2021]. Saatavissa: https://www.zabbix.com/zabbix_agent

[37] Sentry Inc., "Application Monitoring and Error Tracking Software | Sentry," [Viitattu 4.6.2021]. Saatavissa: https://sentry.io/welcome/

[38] Sentry Inc., "Self-Hosted Sentry | Sentry Developer Documentation," [Viitattu 16.4.2021]. Saatavissa: https://develop.sentry.dev/self-hosted/

[39] Django Software Foundation, “Django overview | Django,” [Viitattu 23.4.2020]. Saatavissa: https://www.djangoproject.com/start/overview/

[40] Django Software Foundation, “FAQ: General | Django documentation | Django,” [Viitattu 11.3.2020]. Saatavissa: https://docs.djangoproject.com/en/2.2/faq/general/

[41] V. Driessen, “A successful Git branching model » nvie.com,” [Viitattu 5.2.2021]. Saatavissa: https://nvie.com/posts/a-successful-git-branching-model/

[42] Python Software Foundation, “28.3. venv - Creation of virtual environments - Python 3.6.10 documentation,” [Viitattu 13.3.2020]. Saatavissa: https://docs.python.org/3.6/library/venv.html

[43] Python Software Foundation, “Installing Python Modules (Legacy version) — Python 3.8.9 documentation,” [Viitattu 25.4.2021]. Saatavissa: https://docs.python.org/3.8/install/index.html#modifying-python-s-search-path

[44] Zabbix SIA, “4 Installation from packages [Zabbix Documentation 5.0],” [Viitattu 16.4.2021]. Saatavissa: https://www.zabbix.com/documentation/5.0/manual/installation/

[45] Docker Inc., “Install Docker Engine on CentOS | Docker Documentation,” [Viitattu 16.9.2020]. Saatavissa: https://docs.docker.com/engine/install/centos/

[46] Sentry Inc., “Data Source Name (DSN) | Sentry Documentation,” [Viitattu 17.4.2021]. Saatavissa: https://docs.sentry.io/product/sentry-basics/dsn-explainer/

[47] GitLab Inc., “Run GitLab Runner in a container | GitLab,” [Viitattu 16.9.2020]. Saatavissa: https://docs.gitlab.com/runner/install/docker.html

[48] Django Software Foundation, “Writing and running tests | Django documentation | Django,” [Viitattu 8.4.2021]. Saatavissa: https://docs.djangoproject.com/en/2.2/topics/testing/overview/

[49] “Welcome to the tox automation project — tox 3.23.1.dev12 documentation,” [Viitattu 17.4.2021]. Saatavissa: https://tox.readthedocs.io/en/latest/#what-is-tox

[50] GitLab Inc., “GitLab CI/CD variables | GitLab,” [Viitattu 9.4.2021]. Saatavissa: https://docs.gitlab.com/ee/ci/variables/README.html

[51] Brian Warner, “python-versioneer/INSTALL.md at master · python-versioneer/python-versioneer · GitHub,” [Viitattu 28.4.2021]. Saatavissa: https://github.com/python-versioneer/python-versioneer/blob/master/INSTALL.md

[52] Tom Preston-Werner, “Semantic Versioning 2.0.0 | Semantic Versioning,” [Viitattu 28.4.2021]. Saatavissa: https://semver.org/

[53] “Deploying to Production — pypicloud 1.1.5 documentation,” [Viitattu 22.9.2020]. Saatavissa: https://pypicloud.readthedocs.io/en/latest/topics/deploy.html