Alexios Sarantis

DevOps-työkalujen hyödyntäminen läpi koko elinkaaren

Metropolia Ammattikorkeakoulu

Tekniikan ammattikorkeakoulututkinto

Tieto- ja viestintätekniikan tutkinto-ohjelma

Opinnäytetyö

28.8.2022

Tiivistelmä

Tekijä(t):	Alexios Sarantis
Otsikko:	DevOps-työkalujen hyödyntäminen läpi koko elinkaaren
Sivumäärä:	26 sivua
Aika:	28.8.2022
Tutkinto:	Tekniikan ammattikorkeakoulututkinto
Tutkinto-ohjelma:	Tieto- ja viestintätekniikan tutkinto-ohjelma
Ohjaaja(t):	Janne Salonen

Kiistatta ketterästä kehityksestä on tullut normi IT-alan organisaatioissa ja se on ollut mukana ohjelmistojen toimituksessa yli vuosikymmenen ajan. Kuitenkin Ketterään ohjelmistokehitykseen perustuva DevOps on vielä melko uusi tuttavuus monelle IT-yritykselle. Kun digitaalisia palveluita tuottavien yritysten on nykypäivänä vastattava kysyntään aiempaa nopeammin ja tehokkaammin, juuri DevOps-toimintamalli voi nopeuttaa ohjelmiston kehitykseen, testaukseen ja julkaisuun liittyviä toimintoja automatisoinnin avulla.

DevOpsin tärkeimpänä tavoitteena on kommunikaatio parantaminen työntekijöiden välillä, jotta kaikki työskentelevät tehokkaammin samaa tavoitetta kohti. Tämän uudenalaisen toimintakulttuurin omaksumisen kautta pystytään tuottamaan järjestelmiä nopeammin, tehokkaammin ja laadukkaammin.

Opinnäytetyössä esitellään, miten kehitysprojekti etenee läpi DevOps elinkaaren sen työkaluja hyödyntäen. Työssä avataan keskeisiä käytössä olleita teknologioita ja järjestelmiä, kuten DevOps, jatkuva integraatio, jatkuva toimittaminen, Docker, Git, Gitlab, Jfrog ja PyPI.

Projektin tuloksena prosesseja saatiin automatisoitua, eikä ohjelmistokehittäjiltä enää jatkossa kulu lainkaan aikaa manuaalisten binaarien rakentamineen tai testaamiseen. Lisäksi on tuotu yhteen kaikki tarvittavat tiedot, kuten testitulokset, laatu- ja tietoturva-analyysit, joten katselmoijalle on kaikki tiedot saatavilla palautteen antamiseksi. Tuloksena on syntynyt täysin automatisoitu ratkaisu ja kokonaisuus, joka palvelee ohjelmistokehittäjiä, laadunvarmistustyöntekijöitä ja asiakkaita.

Opinnäytetyö tarjoaa sekä teorian että käytännön osalta hyvän esimerkin DevOps-menetelmien ja välineiden käytöstä ohjelmiston kehityksessä ja julkaisussa. DevOps-toimintamallia voidaan projektin pohjalta laajentaa yrityksen tarpeen mukaisesti.

Avainsanat: DevOps, Ketterä kehittäminen, DevOps elinkaari

Abstract

Author(s):	Alexios Sarantis
Title:	Utilization of DevOps-tools throughout the lifecycle
Number of Pages:	26 pages
Date:	28th of August 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Instructor(s):	Janne Salonen

There is no need to argue that Agile development has become a new normal in IT organizations and Agile methodology has been included in software development for over a decade. However, Agile based DevOps is a somewhat new acquaintance for many IT companies. When companies providing digital services are forced to respond to demand more quickly and efficiently, it is the DevOps operating model that can speed up software development, testing and publishing through automation.

The principal objective of DevOps operating model is to improve communication between workers, so everyone will be working towards the same goal more efficiently. Embracing this new kind of culture, it is possible to produce systems faster, more efficiently and with better quality.

The thesis introduces how the development process proceeds through DevOps lifecycle using DevOps software tools. The thesis defines the fundamental technologies and systems used, like DevOps, continuous integration, continuous deployment, Docker, Git, Gitlab, Jfrog and PyPI.

As a result, the processes were automatized, and the developers no longer need to waste their precious time with manual binary building and testing. Furthermore, all the necessary information, like test results, quality and application security data, are brought together, so the reviewer has all the information at hand. As an accomplishment there is a totally automated solution build up that serves all software developers, quality management and customers.

The thesis provides a good basis for the use of DevOps methods and tools in software development and publishing, both in theory and practice. Furthermore, the DevOps operating model can be expanded according to the company's needs.

Keywords: DevOps operating model, Agile development, DevOps lifecycle

Sisällys

1 Johdanto

Ketterät menetelmät ovat nyt suosittuja kaikkialla, myös IT-alalla, koska niiden ansiosta tiimit ovat joustavia, organisoituneita ja muutokseen reagoivia. Toki ketterä kehittäminen ei ole mitään lastenleikkiä, sen olen työurallani saanut huomata. Jo pelkkä DevOpsin omaksumisen vaatii yritykseltä ketteryyttä ja suoriutumiskykyä, unohtamattakaan sujuvaa kommunikaatiota eri toimijoiden välillä. DevOps on kulttuurinen muutos, joka parantaa kehittäjien ja ylläpitäjien välistä yhteistyötä. Yhdessä oikein käytettynä ketteryys ja DevOps voivat johtaa korkeaan tehokkuuteen ja luotettavuuteen. Ja siihen, että päätösvaltaa suunnataan sinne, missä se tuo parhaan arvon, lähelle asiakasta. (Otaverkko.fi, 2022; Smith, 2011.)

Yritykseni OptoFidelity Oy suunnittelee ja rakentaa automatisoituja testaamiseen tarkoitettuja mittaamisjärjestelmiä mobiili- ja älylaitteiden valmistajille. Palveluntarjoajana sillä on ainutlaatuinen asema maailmassa ja tavoitteena on auttaa asiakkaitaan luomaan maailman parhaat älylaitteet automatisoiden ja digitalisoiden testaamista, ketterästi ja laadukkaasti. Tämän tavoitteen kimpussa olen työskennellyt yrityksessäni DevOps insinöörinä. Onnekseni OptoFidelityllä on ymmärretty, että DevOps lähtee ihmisistä ja ketterä toimintaympäristö on takaamassa tiimien saumatonta toimintaa. (OptoFidelity.com, 2022.)

Projektini käynnistysvaiheessa oli tiedossa, että Jenkins ei enää vastaa yrityksen nykypäivän tarpeisiin. Työlleni oli siis selkeästi osoitettu tarve ja tavoitteena oli saada binary paketointi ja python package indexit samaan paikkaan, Jfrog Artifactoryyn ja hyödyntää Gitlab CI:ta niin, että yritys voisi luopua Jenkinsistä. GitLab CI on selkeästi soveltuvampi valinta yritykselleni, koska se oli jo osa käytössä olevaa versionhallintajärjestelmää ja voisimme näin ollen jatkossa hyödyntää kaikkia GitLabin ominaisuuksia. Opinnäytetyössäni esittelen, miten projektini eteni läpi DevOps elinkaaren sen työkaluja hyödyntäen. Avaan myös keskeisiä käytössä olleita teknologioita ja järjestelmiä.

2 Keskeiset käsitteet ja teknologiat

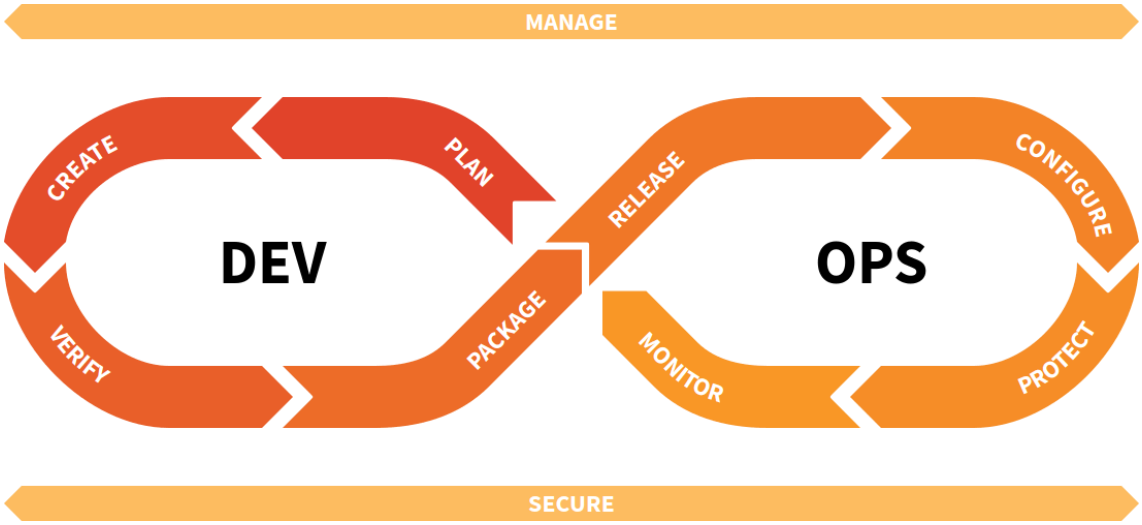
2.1 DevOps

Insinöörityöni käsitteistä ja teknologioista ehkä vaikein avata on DevOps. Periaatteessa se on hyvin yksinkertainen asia. DevOps toimii ohjelmistokehityksen ”development” ja palvelutarjonnan ”operations” eli tuotantoon viennin, operoinnin ja monitoroinnin automatisoinnin välineenä. Tässä toimintamallissa kehityksen, testaamisen ja ylläpidon toiminnot tehdään valmiin sapluunan mukaan, automatisoidaan mahdollisimman pitkälle ohjelmistokehityksen, testaamiseen ja ylläpidon toiminnot. Tuloksena syntyy paremmin koordinoitu ja entisiä ketterämpi työtapa kehityksen ja palveluoperoinnin välille. DevOps on yhdistelmä työkaluja, ihmisiä ja kulttuuria. DevOpsin filosofiaan sisältyy jatkuvan oppimisen, kommunikaation ja kehittämisen kulttuuri ja se on tullut jäädäkseen osaksi IT-alan kehitysprojekteja. (Abildskov, 2021; Kim, 2016.)

Tärkeintä on ymmärtää, että DevOpsia ei voi ostaa valmiina tai saada asentamalla DevOps-nimellä mainostettuja tuotteita, sillä ne eivät yksi tee Devopsia. Yläkäsitteenä DevOps toki kattaa laajasti toimintoja aina insinöörityöstä tuotehallintaan. DevOpsin ehdottomana etuna on nopea ja luotettava ohjelmiston rakentaminen, testaaminen ja julkaisu. Toimintamallin parasta antia on loppukäyttäjiltä saatu palaute, joka saadaan tuotantoympäristöistä kehitystiimille ja tiimit voivat näin reagoida markkinaolosuhteisiin nopeammin ja päihittää kilpailijat tehokkaalla toiminnalla, joka läpäisee koko organisaation. DevOpsiin juurtunut jatkuvan oppimisen ja kokeilun kulttuuri tuottaa asiakkaille mahdollisimman paljon arvoa tehokkaasti ja riskittömästi. (Abildskov, 2021; Kim, 2016.)

DevOps elinkaari (lifecycle, kuva 1.) hallitsee kehitysryhmän ja operaatiotiimin välistä suhdetta, jotta oikea tuote voidaan asentaa asiakkaalle nopeammin automaatiotyökaluja hyväksi käyttäen. Elinkaari koostuu eri vaiheista ja sen standardoitu, mutta joustava rakenne ohjaa käyttäjää prosessin eri vaiheesta toi-

seen. Elinkaaren seitsemän vaihetta ovat kehittäminen (development), integraatio, testaus, käyttöönotto (deployment), monitorointi, palaute (feedback) ja (operations). Elinkaarimalli osoittaa DevOpsin yhden tärkeimmistä tavoitteista: purkaa yhden osaajan rakenteita yrityksessä. Kommunikaation ja yhdessä tekemisen kulttuuri kehitystyössä mahdollistaa sen, että kaikki ei kaadu sen yhden osaajan ollessa poissa vaan kaikilta tiimistä löytyy ymmärrys ja tieto siitä, mitä on tehty ja ollaan tekemässä. (Gitlab docs, 2022.)



Kuva 1. DevOps elinkaari eli life cycle (Gitlab docs, 2022).

2.2 Jatkuva integraatio - Continuous Integration

Jatkuva integraatio (Continuous Integration, CI) tarkoittaa toimintatapaa, jota tukee valikoima työkaluja ja ohjelmistoja ja jossa sovelluskehitystiimin tuottamat muutokset versiohallintaan todennetaan mahdollisimman nopeasti ja automaattisesti. Näin varmistetaan niiden laatu ja se, että ne eivät esimerkiksi riko ohjelmistoa tai ohjelmistokappaletta. (Meyer, 2014.)

Jatkuva integraatio on tärkeä toimintatapa DevOps elinkaaren mukaisessa kehityksessä, sillä nopean kehityssyklin edellytyksenä on nopea palaute muutoksista. Jatkuvan integraation prosessi lähtee alkuun siitä, että ohjelmistokehittäjä

vie versiohallintaan haluamansa koodimuutoksen, jonka jälkeen jatkuvaan integraatioon erikoistunut palvelin hakee koodimuutokset, ajaa valikoiman yksikkö-, lisenssi- ja tietoturvatestejä, mittareita ja muita työkaluja, joiden lopputuloksena syntyy raportteja ja tieto siitä, onko paketointi (engl. build) onnistunut. Jatkuvan integraation myötä ohjelmistokehittäjä saa reaaliaikaisen tiedon siitä, onko hänen tekemänsä testimuutos sellainen, että se soveltuu tuotantoon asti ilman, että se hajottaa mitään olemassa olevaa. (Smeds et al., 2015; Meyer, 2014.)

Jatkuvan integraation alustoja on nykyisin useita, ja ne ovat useammin integroitu osaksi versiohallinnan kokonaisuutta, tästä esimerkkinä Gitlab CI, joka toimii osana Gitlab versiohallintaa. Gitlab CI on valikoitunut jatkuvan integraation palvelimeksi myös tässä projektissa.

2.3 Jatkuva toimittaminen - Continuous Deployment

Jatkuvalla toimittaminen (Continuous Deployment, CD) tarkoitetaan nimen mukaisesti ohjelmiston jatkuvaa toimittamista tuotantoon. Jatkuva toimittaminen ja jatkuva integraatio toimivat käsi kädessä. Jatkuva toimittaminen vie muutokset tuotantoympäristöön heti sen jälkeen, kun jatkuva integraatio on varmistanut, että kaikki muutokset ja prosessit ovat menneet läpi onnistuneesti. (Leppänen et al., 2015.)

Näin ollen jatkuva toimittaminen tarkoittaa ohjelmiston automatisoitua asennusta tuotannon ympäristöön. Tähän kuuluu useita osioita, joista ohjelmiston asennus vain yksi, mutta toki jatkuvalla toimittamisella voidaan hallita esimerkiksi infrastruktuurin konfiguraatiota. Keskeisimpänä hyötynä jatkuvalle toimittamiselle, kuten DevOpsillekin, on nopea palaute loppukäyttäjältä. Tämän lisäksi hyödyllistä on, että ohjelmistokehitystiimi voi itse hallita suoraan, milloin he vievät muutokset tuotantoon tai ottavatko niitä pois tuotannosta. (Leppänen et al., 2015.)

2.4 Docker

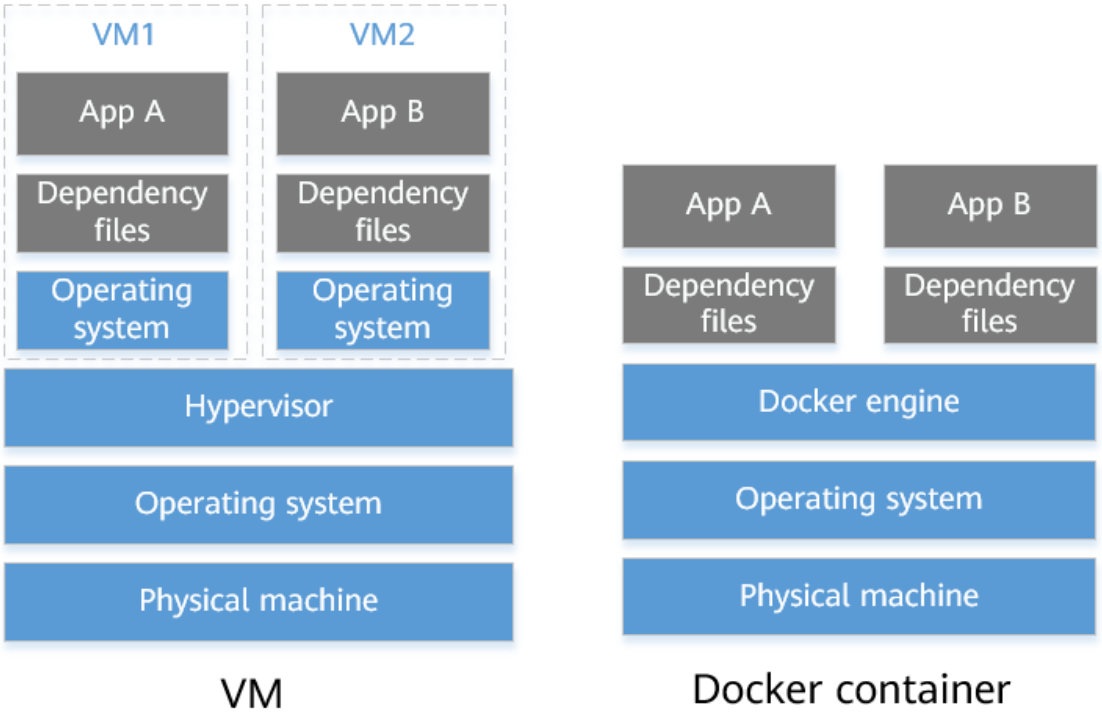
Docker on konttiteknologia, jota käytetään paljon erilaisten projektien tai sovellusten ajossa. Se perustuu avoimeen lähdekoodiin perustuvaan ohjelmistoon, jota käytetään sovelluksen ja sen käyttämien sovelluskirjastojen ja komponenttien paketoimiseen. Dockerin avulla voidaan luoda tilanne, jossa sovellus saadaan asennettua aina samoin vaatimuksin ja riippuvuuksin kohdekoneelle, sovellus paketoidaan Docker-konttiin. Kontissa on yleensä Linux-käyttöjärjestelmä käytössä ilman Linux kerneliä. Toisin sanoen käyttöjärjestelmä hyödyntää kohdekoneen resursseja, mutta ottaa näistä vain määrätyn osan. (Chamberlain, 2018; Docker docs, 2022.)

Docker-kontin perustana ovat Docker imaget. Ne koostuvat useammasta sovelluskerroksesta (layer) ja yhdessä kerroksessa kuvataan yksi Docker-tiedosto (Dockerfile). Yhteen Docker imageen voidaan paketoida yksittäisiä komponentteja ja se sisältää käyttöjärjestelmän kuten Window tai Linux muodostaen sovelluskerroksia. Docker image on näin ollen kuin täytekakku eri kerroksineen: yksi komponentti on hillo, toinen marjat, kolmas kermavaahto ja käyttöjärjestelmä on kakkupohja. Docker image luodaan ns. base imagen päälle, ja se sisältää käyttöjärjestelmän. Docker image on muuttumaton ja sillä ei ole määriteltyä tilaa (state). Tämän uudelleen käytettävyytensä ansiosta se on tehokkaasti hyödynnettävissä ohjelmistokehityksessä ilman, että on pelkoa alustan rikkoutumisesta. (Docker docs, 2022.)

Docker-kontti (docker container) on Docker imagen pohjalta ajettava varsinainen instanssi eli sovelluksen ajoympäristö. Kontille ei aseteta etukäteen määrättyjä resursseja, vaan Docker osaa pyytää isäntätietokoneen käyttöjärjestelmältä resursseja tarpeen mukaan. Kaikki kontin luomisen jälkeen tehtävät muutokset tehdään kontin sisälle. (Docker docs, 2022.)

Docker-rekisteriä käytetään Docker-imagejen tallentamiseen ja levittämiseen ja rekisterin kautta on mahdollista ladata valmiita Docker-imageja tai tallentaa

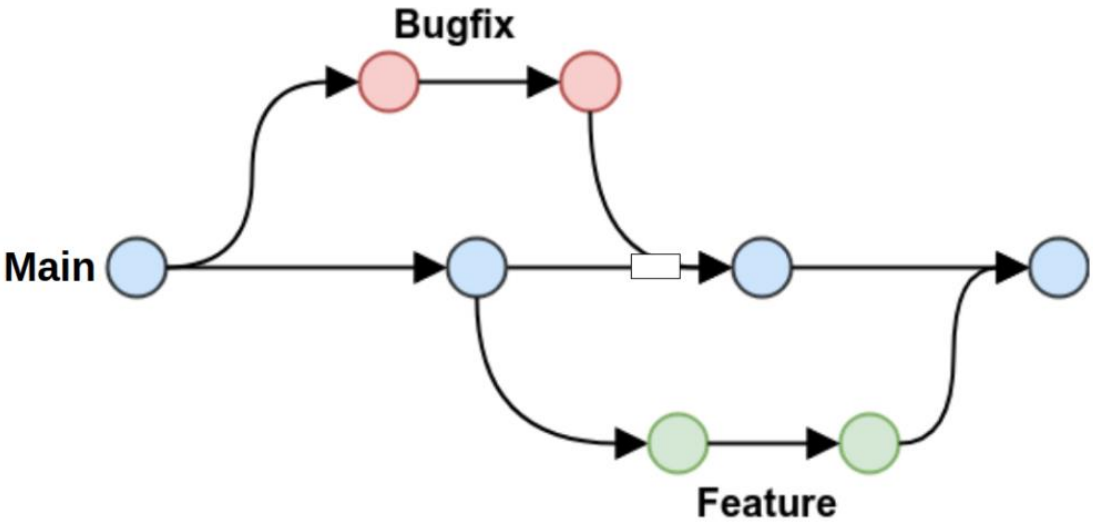
omia imageja. Docker voi käyttää Docker Hub -rekisteriä, jonka kautta on ladattavissa valmiita Docker-imageja. Vaihtoehtoisesti Docker-rekisteri voi olla myös omassa pilvipalvelussa, kuten yrityksessäni on. (Mouat, 2016.)



Kuva 2. Docker vs. Virtual Machine (Docker docs, 2022).

2.5 Versiohallintajärjestelmä Git

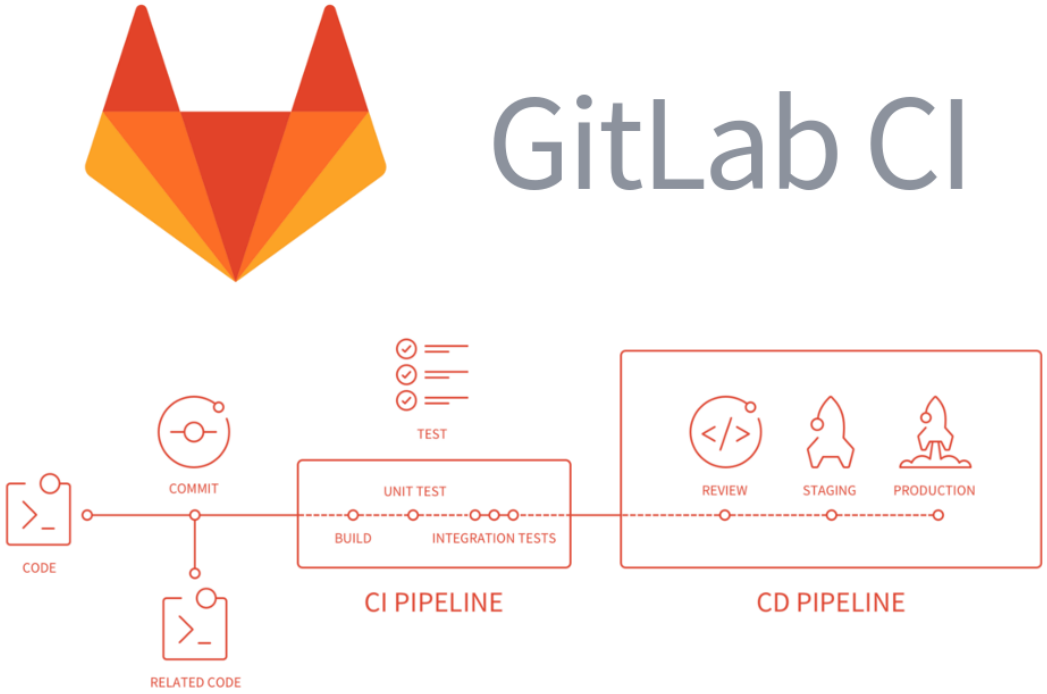
Git on versionhallintajärjestelmä Linux-ytimen kehittämisprojektille, jonka Linus Torvaldsin ja Linuxin kehittäjät ovat luoneet. Git on suunniteltu varmistamaan datan yhtenäisyys ja se, että koodimuutokset eivät päivitä vain osaa muutoksista virheen sattuessa. Tämä on elintärkeää ympäristössä, jossa datan yhtenäisyydellä on suuri merkitys. (Chacon, 2014; Somasundaram, 2013.)



Kuva 3. Git prosessi (Github.com, 2022).

2.6 Gitlab ja Gitlab CI

GitLab on perustettu vuonna 2011 ja sillä oli DevOps alustan ansiota yli 30 miljoonaa rekisteröityä käyttäjää vuonna 2021. GitLab on suunnattu ominaisuuksiltaan enemmän suurille yrityksille kuin kilpailijansa. Se antaa mahdollisuuden käyttäjäryhmille, joiden perusteella voidaan antaa oikeudet eri projekteihin tai sen osa-alueisiin. Tässä projektissa jatkuvan integroinnin työkaluksi valittiin GitLab CI. Se on avoimen lähdekoodin työkalu, joka integroituu erittäin hyvin osaksi GitLabia. (Gitlab docs, 2022.)



Kuva 4. GitLab CI prosessi (Gitlab.com, 2022).

2.6.1 Gitlab runnerit

Runner on applikaatio, joka toimii Gitlab CI kanssa ja suorittaa tehtävät pipelinessa. Se voi olla Docker-image, virtuaalikone tai fyysinen kone. Runnereita on kahdenlaisia, shared eli jaettuja ja specific eli projektikohtaisia. Jaetut runnerit ovat automaattisesti skaalautuvia. Skaalaus vähentää odotusaikoja ennen töiden aloitusta ja se mahdollistaa jokaisen projektille eristetyn virtuaalikoneen käytön. (Gitlab docs, 2022.)

2.7 Jfrog artifactory

JFrog Artifactory on universaali DevOps-ratkaisu. Se on tietovarasto, joka hallinnoi pakettitiedostoja, Docker-kontteja, komponentteja ja tiedostoja. Jfrog Artifactory palvelee keskitetysti fyysisiä tietoikoneita ja virtuaalikoneita, kun halutaan hakea tarvittavia tiedostoja. (Jfrog.com, 2022.)

2.8 Python Package Index (PyPI)

Python Package Index lyhyesti PyPI työkalua käytetään Pythonissa kirjoitettujen pakettien asentamiseen ja hallintaan (Pypi.org, 2022).

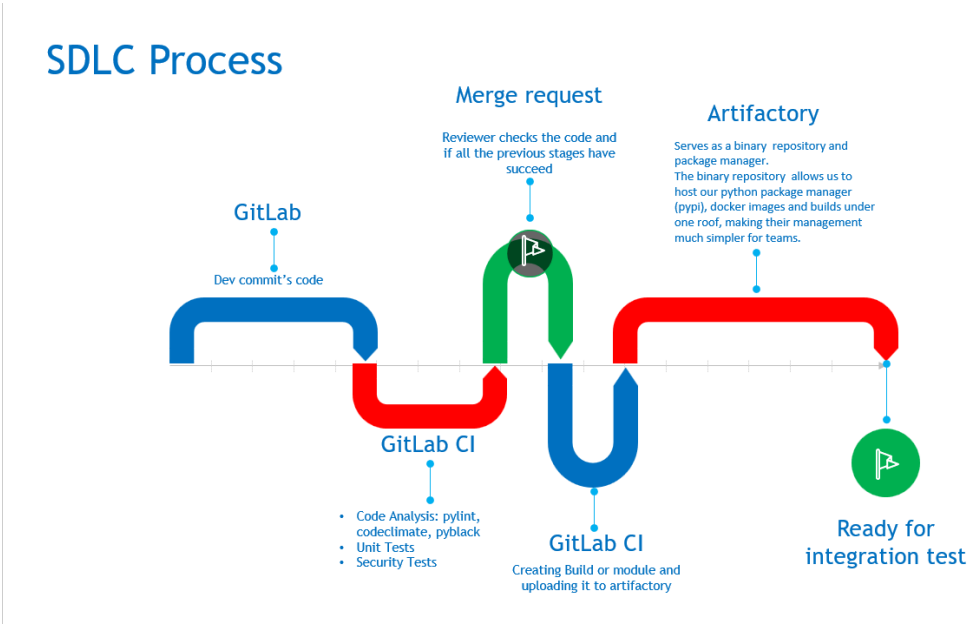
3 Insinöörityön tavoitteet ja tarkoitus

OptoFidelity käyttää jatkuvaan integraatioon ja jatkuvaan toimittamiseen (CI/CD) Jenkins työkalua ja versionhallintajärjestelmänä käytössä on GitLab. Lisäksi heillä on erikseen oma virtuaalikone, johon on pystytetty Python package index (Pypi).

Tämän projektin tavoitteena on saada kaikkien OptoFidelityn projektien binary paketoinnit ja Python package index samaan paikkaan, Jfrog Artifactoryyn. Lisäksi pyritään hyödyntämään GitLab CI:a niin, että voidaan päästä eroon Jenkinsistä. Muutos on perusteltu, koska OptoFidelityssä koetaan, että Jenkins ei vastaa yrityksen nykypäivän tarpeisiin. GitLab CI on soveltuvampi valinta yritykselle, koska se on osa jo käytössä oleva versionhallintajärjestelmää ja voi näin ollen hyödyntää kaikkia GitLabin ominaisuuksia.

Projektini tarkoituksena on automatisoida prosesseja niin, että kehittäjien ei tarvitse käyttää aikaa manuaalisen binaarin rakentamiseen tai testaukseen. Näin säästyy paljon aikaa ja rahaa yrityksessä sekä asiakkailla. Tuloksena syntyy toimiva ratkaisu ja kokonaisuus, joka on täysin automatisoitu.

Kuvassa 5. olen esitellyt, miten projektini etenee SDLC-mallin mukaisesti. Ohjelmistokehityksen elinkaaren (SDLC) alussa kehittäjä työntää (git push) kehittämänsä koodia omalle haaralle (branch) ja sen seurauksena Gitlab CI käynnistyy, analysoi koodia ja testaa sitä. Seuraavassa vaiheessa (merge request) katselmoija(t) tarkistaa, mitä muutoksia koodissa on tehty ja onko kaikki testit, analyysit menneet läpi. Kun katselmoija on hyväksynyt merge requestin, GitLab CI aktivoituu ja aloittaa buildin tai moduulin muodostamisen. Paketoinnit viedään Gitlab CI kautta suoraan tietovarastoon ja moduulit suoraan python package indexille. Artifactoryyn päätyneet paketoinnit ja moduulit ovat testattavissa laadunvarmennustiimin toimesta (Integration test).



Kuva 5. SDLC Ohjelmistokehityksen elinkaari projektissa.

4 Toteutus ja tulokset

4.1 Projektin toteutus

Lähdin projektityössäni liikkeelle Devops elämänkaaren vaiheiden mukaisesti: etenin kehittämisestä integraation kautta testaukseen ja edelleen käyttöönottoon.

Projektin alkuvaiheessa otettiin käyttöön Gitlab runnerit. Ne asennettiin (7 kpl) virtuaalikoneisiin, jotka tarjoaa ulkopuolinen palveluntarjoaja. Runnereista asennettiin neljä Windows- ja kolme Linux-virtuaalikoneisiin, koska tietyt ohjelmistot yrityksessä toimivat vain Windows ympäristössä. Runnereiden toteuttajaksi (executor) määriteltiin Docker.

GitLab CI:n käyttöönotto tapahtui luomalla tiedosto .gitlab-ci.yml GitLab-projektin juureen. Tämä tiedosto on YAML-formaatissa ja se ohjaa GitLab CI:ta siinä, kuinka työ pitäisi tehdä.

```
#####################################################
#                 ### PYTEST ###                    #
#####################################################

Unit tests:
  stage: test
  script:
    - python3 -m venv venv
    - source venv/bin/activate
    - cp ${PIP_REPOSITORY_CONFIG} venv/pip.conf
    - python3 -m pip install --upgrade ".[test,bundle]"
    - apt-get update
    - apt-get install -y libgl1
    - python3 -m pytest ./tests  --junitxml=${CI_PROJECT_DIR}/report.xml
    - coverage xml -o cov.xml -i
  coverage: '/TOTAL.*\s([.\d]+)%/'
  artifacts:
    when: always
    reports:
      junit: ${CI_PROJECT_DIR}/*.xml
      coverage_report:
        coverage_format: cobertura
        path: cov.xml
```

Kuva 6. .gitlab-ci.yml esimerkki.

Projektin seuraavassa vaiheessa aloin kartoittaa, mitä teknisiä riippuvuuksia tai esteitä yrityksen projekteilla oli. Tämän kartoituksen myötä pääsin luomaan Docker imageja, jotka vietiin Jfrog artifactoryyn käytettäväksi GitLab CI:ssa. Linux Docker imageihin käytin Docker hubin tarjoamia valmiita imageja. Windows Docker imageiden luominen oli haastavaa, koska riippuvuudet ovat kooltaan aika isoja, joten riippuvuudet piti integroida osaksi imagea. Joten imageihin lisättiin Pythonin eri versioita, Git, Powershell core, Innosetup, Visualstudio sekä kaikki ohjelman laajennukset, Cmake ja Chocolatey.

Koska yrityksessä oli erikseen oma virtuaalikone, johon oli pystytetty python package index (Pypi), siirsin (migration) paketit uuteen package indexiin, joka on pystytetty Jfrog artifactoriin. Tämä työni vaihe ei ollut kaiken kaikkia yksinkertainen, aikaa veivät erityisesti Windows Docker imaget, sillä aina ilmeni ennalta arvaamattomia riippuvuuksia.

dockerfile 3.45 KiB Edit Re

```
# escape=`
FROM mcr.microsoft.com/windows:20H2

SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]

# https://github.com/docker-library/python/pull/557
ENV PYTHONIOENCODING UTF-8

ENV PYTHON_VERSION 3.9.7
ENV PYTHON_RELEASE 3.9.7

RUN $url = ('https://www.python.org/ftp/python/{0}/python-{1}-amd64.exe' -f $env:PYTHON_RELEASE, $env:PYTHON_VERSION); `
        Write-Host ('Downloading {0} ...' -f $url); `
        [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12; `
        Invoke-WebRequest -Uri $url -OutFile 'python.exe'; `
        `
        Write-Host 'Installing ...'; `
# https://docs.python.org/3/using/windows.html#installing-without-ui
        $exitCode = (Start-Process python.exe -Wait -NoNewWindow -PassThru `
                -ArgumentList @( `
                        '/quiet', `
                        'InstallAllUsers=1', `
                        'TargetDir=C:\Python', `
                        'PrependPath=1', `
                        'Shortcuts=0', `
                        'Include_doc=0', `
                        'Include_pip=0', `
                        'Include_test=0' `
                ) `
```

Kuva 7. Dockerfile 1/2.

```
        Write-Host 'Complete.'

SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference = 'SilentlyContinue';"]
RUN Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor
RUN choco install -y pwsh
RUN cinst -y git
RUN choco install -y innosetup
RUN choco install visualstudio2019-workload-vctools --package-parameters "--includeRecommended" -y
RUN choco install cmake -y
```

Kuva 8. Dockerfile 2/2.

Kun Docker imaget ja Python paketit olivat valmiina käytettäviksi, aloitin lisäämään töitä GitLab CI:hin. Koodin ja tyylin analysointiin valitsin käytettäväksi Co-

declimate, Pylint ja Python black ohjelmat, Pylintiä varten tein myös oman scriptin, joka tarkistaa koodin muokatut ja uudet, lisätyt tiedostot ja antaa niille erikseen arvosanan ja palautteen kehittäjälle. Pytest ajaa kehittäjien itse luomia yksikkötestejä. Näiden kaikkien töiden tulokset/raportit liitetään automaattisesti merge requestiin.

```
################################################
#               ### PYLINT ###                 #
################################################
# Run pylint for modified files.
# By forcing high(ish) score the overall code quality should increase over time.
# The threshold shall be lowered if it's too difficult to fulfill the requirements.
pylint-changed-files:
  stage: lint
  extends: .installing-files
  script:
    - >
      if [ `git diff --name-only --diff-filter=M $CI_DEFAULT_BRANCH src | grep ".*.py" | wc -l` -ge 1 ]; then
      python3 -m pylint `git diff --name-only --diff-filter=M $CI_DEFAULT_BRANCH src | grep ".*.py"` --fail-under=8;
      fi
  allow_failure: true

################################################
#               ### PYLINT ###                 #
################################################
# Run pylint for modified files and their previous versions.
# This step requires the modified files to have pylint score higher or equal to the score before changes.
pylint-score-increase-check:
  stage: lint
  extends: .both
  after_script:
    - bash lint_modified.sh

################################################
#               ### PYLINT ###                 #
################################################
# run pylint against python sources
# overrides default before_script
pylint:
  stage: lint
  before_script:
    - PIP_CONFIG_FILE=${PIP_REPOSITORY_CONFIG} python3 -m pip install --user --upgrade pylint
    - PIP_CONFIG_FILE=${PIP_REPOSITORY_CONFIG} python3 -m pip install --user --upgrade .
  script:
    - python3 -m pylint src --fail-under=5
  allow_failure: true #Use allow_failure only if you have permission
```

Kuva 9. Pylint script .gitlab-ci.yml tiedostossa.

```
#!/usr/bin/env bash

NEWSCORES=()
OLDSCORES=()
FILENAMES=`git diff --name-only --diff-filter=M master src | grep ".*.py"`
EXITCODE=0
INDEXABLEFILENAMES=()

touch tempfile.py

for FN in ${FILENAMES[@]}; do
  INDEXABLEFILENAMES+=( $FN )
done

for FN in ${FILENAMES[@]}; do
  NEWSCORES+=( `python3 -m pylint $FN | sed -n 's/^Your code has been rated at \([-0-9.]*\)\/.*/\1/p'` )
done

for FN in ${FILENAMES[@]}; do
  git show origin/$CI_DEFAULT_BRANCH:${FN} > tempfile.py
  OLDSCORES+=( `python3 -m pylint tempfile.py | sed -n 's/^Your code has been rated at \([-0-9.]*\)\/.*/\1/p'` )
done

for I in ${!NEWSCORES[@]}; do
  if `python3 -c "exit(0 if ${NEWSCORES[I]} < ${OLDSCORES[I]} else 1)"`; then
    echo "New score lower than old in file ${INDEXABLEFILENAMES[I]}. New score: ${NEWSCORES[I]} Old score: ${OLDSCORES[I]}"
    EXITCODE=1
  fi
done

rm tempfile.py
exit $EXITCODE
```

Kuva 10. Pylint bash script.

Seuraavassa vaiheessa ajetaan GitLab CI:ssa compliance testit. GitLab ultimate lisenssi tarjoaa tämän ominaisuuden, mutta OptoFidelityllä lisenssiä ei ollut. Tästä syystä loin omat scriptit/testit, jotka käsittelevät ja tarkistavat kaikki asennetut Python moduulit ja niiden lisenssit. Tiettyjä lisenssejä kuten GPL (General Public License) ei ole sallittua käyttää yrityksessämme paketoinnin yhteydessä.

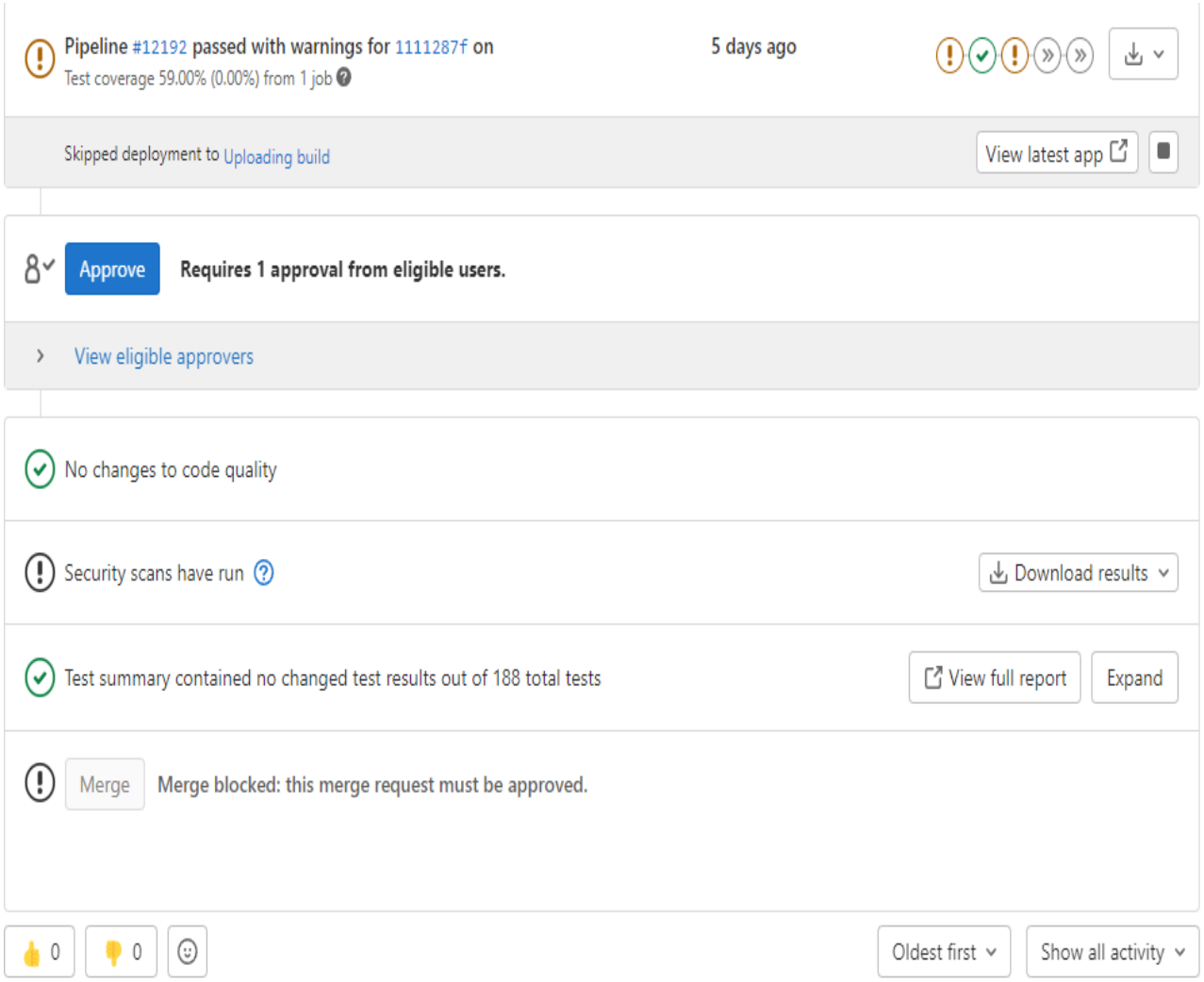
```
import sys
import importlib.metadata

from robot.api import logger

class TestFailedError(Exception):
    """Created error for failed tests"""

class DeprecationError(Exception):
    """Created error for failed tests"""

# The constant lists at the top of the file to make them easy to modify.
BLACKLISTED_DISTRIBUTIONS = [
    "vboxapi",
    "example1"
    # Add blacklisted distributions here
    # ...
]

DEPRECATED_DISTRIBUTIONS = {
    # deprecated: replacement
    "numpy-old": "numpy-new",
}

BAD_LICENSES = [
    "GPL",
    "GPL v3",
    "LGPL v1",
    "LGPL v2",
    "LGPL v3",
]
BAD_LICENSE_WHITELIST = [
    "PyQt5",
    "PyQt5-Qt5",
    "PyQt5-sip",
    # ...
]
```

Kuva 11. Compliance script 1/2.

```
@keyword
def compiliance_test():
    """Checking installed modules if they have unsupported licenses"""
    python_version_checker()
    notallowed = 0
    logger.console("")
    for pkg in importlib.metadata.distributions():
        if (
            pkg.metadata.get("License") in BAD_LICENSES
            and pkg.metadata.get("name") not in BAD_LICENSE_WHITELIST
        ):
            logger.console(
                "ERROR: Package {} has unsupported license {}!".format(
                    pkg.metadata.get("name"), pkg.metadata.get("License")
                )
            )
            notallowed += 1
    if notallowed > 0:
        raise TestFailedError("Unsupported license(s) found")
    else:
        logger.console("Only supported licenses found")
```

Kuva 12. Compliance script 2/2.

Kun työt ovat läpäisseet kaikki edelliset vaiheet ja merge request hyväksytty, seuraavassa pipeline työssä luodaan projektien binaarit tai paketteja ja niitä viedään automaattisesti Jfrog artifactorylle. Yleensä binaarit luodaan Windows ympäristössä, mutta esimerkiksi kun viedään paketteja tai valmiita binaareita artifactorylle, käytetään Linux ympäristöä nopeuden vuoksi. Projektin tässä vaiheessa ei ilmennyt vastoinkäymisiä ja kaikki sujui suunnitelmien mukaisesti aikataulussa. Pipelinen pystyttämiseen kului ajallisesti noin kaksi viikkoa



Kuva 13. Projektin tuloksena muodostunut merge request.

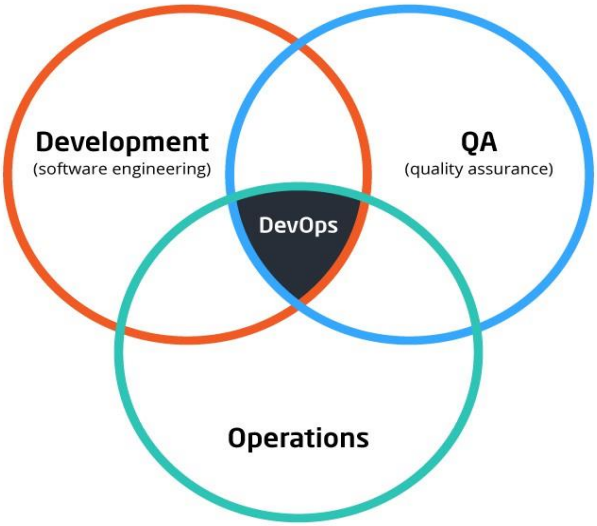
4.2 Tulokset ja palaute

Projektini tavoitteena oli saada kaikkien OptoFidelityn projektien binary paketoinnit ja Python package index samaan paikkaan, Jfrog Artifactoryyn. Onnistuin saavuttamaan tavoitteeni ilman suurempia vastoinkäymisiä sille varatussa ajassa. Työni ansiosta saimme yritykseni prosesseja automatisoitua, eikä ohjelmistokehittäjiltä enää jatkossa kulu lainkaan aikaa manuaalisten binaarien rakentamineen tai testaamiseen. Merge requestille on tuotu kaikki tarvittavat tiedot, kuten testitulokset, laatu- ja tietoturva-analyysit, joten katselmoijalle on kaikki tiedot saatavilla palautteen antamiseksi. Tuloksena on siis syntynyt täysin automatisoitu ratkaisu ja kokonaisuus, joka palvelee ohjelmistokehittäjiä, laadunvarmistustyöntekijöitä ja asiakkaita.

Toki vasta tulevaisuudessa (vuonna 2023) projektin osalta on vuorossa monitorointi, mutta sitä ennen odotan jo saavani paljon jatkuvaa palautetta tehdystä kehittämistyöstä. Jatkuva palaute mahdollistaa työni edelleen kehittämisen ja jatkuvan ylläpidon.

Jatkossa yrityksessä on tarkoitus asentaa GitLab runner testirobotille ja tehdään integraatiotestaus osana jatkuvaa integraatiota. Tällä hetkellä yrityksessä integraatiotestaus tehdään vielä manuaalisesti.

Projektin myötä olen oppinut paljon uusista teknologioista ja työkaluista, ja tästä on minulle paljon hyötyä tulevalla työurallani. Tässä roolissa, DevOps insinöörinä olen nauttinut työni monipuolisuudesta, olen ohjelmistokehittäjä, testaaja ja data-analyytikko. Toimintakulttuurini yrityksessämme mahdollistaa nykyisellään hienosti DevOps menetelmien hyödyntämisen, mutta silti kulttuurin ylläpitämisen ja kehittämisen eteen on jatkuvasti tehtävä työtä. On muistettava pitää yksilöt ja vuorovaikutus prosessien ja työkalujen edellä ja valmistauduttava reagoimaan jatkuvaan muutokseen ja tavoitteiden uudelleen arviointiin tilanteen niin vaatiessa.



Kuva 14. DevOps insinöörin tehtävät.

Lähteet

Abildskov, J. 2021. Mitä on DevOps? Luettu 26.5.2022. https://www.eficode.com/fi/blog/mita-on-devops

Chamberlain, D. 2018. Containers vs. Virtual Machines (VMs): What's the Difference? Luettu 10.8.2022. https://blog.netapp.com/blogs/containers-vs-vms/

Chacon, S. & Straub, B. 2014. Pro Git. Apress.

DevOps prosessi. 2022. Luettu 20.07.2022. https://about.gitlab.com/topics/devops/

Docker docs. Luettu 25.07.2022. https://docs.docker.com/

Gitlab docs. Luettu 24.07.2022. https://about.gitlab.com/company/history/

JFrog documentation. 2022. Luettu 14.8.2022. https://www.jfrog.com/confluence/display/JFROG/JFrog+Documentation

Kim, G. et al. 2016. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. Portland: IT Revolution Press.

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V., Itkonen, J., Mäntylä, M. V. & Männistö, T. 2015. The highways and country roads to continuous deployment. IEEE software, 32(2), 64-72.

Meyer, M. 2014. Continuous Integration and Its Tools. IEEE Software 31(3), 14– 16.

Mouat, A. 2016. Using Docker. O´Reilly. Luettu 26.08.2022. https://store.dockerme.ir/E-Book/Using_Docker.pdf

OptoFidelity.com. 2022. About OptoFidelity. Luettu 14.8.2022. https://www.optofidelity.com/company

Otaverkko.fi. 2022. Mitä on DevOps? Luettu 14.8.2022. https://otaverkko.fi/mita-on-devops/

Python Packege Index. 2022. Luettu 5.8.2022. https://pypi.org/

Smeds, J, Nybom, K & Porres Paltor, I. 2015. DevOps: A Definition and Perceived Adoption Impediments. Helsinki.

Smith, DM. 2011. Hype cycle for cloud computing. Gartner Inc. Luettu 20.8.2022. https://cmapspublic3.ihmc.us/rid=1JZJKBR35-2C5G28T-ZNM/hype_cycle_for_cloud_computi_214915.pdf

Somasundaram, R. 2013. Git: Version Control for Everyone. Birmingham: Packt Publishing.