



Jensina Hakkarainen

Jatkuvan integraation ratkaisut Azure DevOps -ympäristössä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

26.2.2023

Tiivistelmä

Tekijä:	Jensina Hakkarainen
Otsikko:	Jatkuvan integraation ratkaisut Azure DevOps -ympäristössä
Sivumäärä:	47 sivua
Aika:	26.2.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander Sovellusarkkitehti Joni Sekki

Insinööriyön tavoitteena oli luoda toimeksiantajayritys Oy Samlink Ab:lle jatkuvan integraation putki erääseen rahoitustenhallintajärjestelmän projektiin. Putki toimisi validointina pull requesteille ja sen tavoitteena oli tuoda laadunhallintaa varhaisempaan kehitysvaiheeseen, sillä aiemmin yksikkötestaus ja koodin laadun varmistus olivat kehittäjien subjektiivisen muistin ja osaamisen varassa.

Insinööriyössä pohdittiin ohjelmiston ja koodin laatua ja määrittelyä ja niistä ammennettiin yleisten hyvien käytäntöjen mukaisia sääntöjä ja ehtoja putken läpimenoille. Putki toteutettiin hyödyntäen Azure Pipelines -palvelua. Putkessa oli alkuperäisen suunnitelman mukaan oltava koonti, yksikkötestaus, SonarQube-analyysi, WhiteSource-analyysi sekä testikattavuusraportti, jonka myötä saataisiin asetettua 80 prosentin vähimmäisvaatimus ehdoksi putken läpimenoille ja pull requestin hyväksynnälle.

Projektia kehitettiin Azure DevOps -palvelimella. Vanha koodi oli toteutettu .NET Framework 4.7.2 -ohjelmistokehyksellä ja uusi koodi .NET 6 -ohjelmistokehyksellä. Vanha 4.7.2-ohjelmistokehys aiheutti ongelmia testikattavuustyökalujen kanssa, sillä useimmat niistä eivät tue kyseistä kehystä. Lisäksi testikattavuustyökalun valintaan vaikutti SonarQube-yhteensopivuus. Tämän opinnäytetyön valmistumishetkellä projektilla ei ollut käytössä testikattavuusanalyysia, mutta testiprojektiin tämä saatiin toimimaan .NET 6 -ohjelmistokehyksen kanssa. Lopputuloksena syntyi putki, joka sisälsi koonnin, yksikkötestauksen sekä SonarQube-analyysin. Jatkokehityksen kohteena on löytää toimiva testikattavuusratkaisu, joka toimii molempien ohjelmistokehysten kanssa.

Avainsanat: Microsoft Azure, Azure DevOps, Azure Pipelines, Jatkuva integraatio, SonarQube, Ohjelmiston laatu, Koodin laatu

Abstract

Author: Jensina Hakkarainen
Title: Continuous Integration Solutions in Azure DevOps
Number of Pages: 47 pages
Date: 26th February 2023

Degree: Bachelor of Engineering
Degree Programme: Information and Communication Technology
Professional Major: Software Engineering
Supervisors: Simo Silander, Senior Lecturer
Joni Sekki, Software Architect

The goal of the study was to create a continuous integration pipeline for Oy Samlink Ab's financial management product. The pipeline works as a pull request validation and the aim was to bring quality assurance to an earlier stage of the development. Before the responsibility of unit testing and quality assurance rested on the developers' subjective memory and knowledge.

This thesis studies software and code quality and their definitions. The policies and conditions for the pipeline to pass were gathered from the common best practices of software and code quality. The pipeline was done with the Azure Pipelines service. In the original plan the pipeline was to consist of build, unit testing, SonarQube analysis, WhiteSource analysis and code coverage analysis, which would allow to add 80 percent minimum test coverage requirement for the validation pipeline to pass and the pull request to be accepted.

The project was developed in the Azure DevOps server. Legacy code was done in .NET Framework 4.7.2 and new code in .NET 6 framework. The legacy code caused issues with the code coverage tools since most of the tools did not support the old 4.7.2 framework. Another thing to consider when choosing the tool was SonarQube compatibility. On the moment of writing the thesis the project does not have the code coverage analysis, but it currently works in the .NET 6 test environment. WhiteSource was also left out of the pipeline due to the service not supporting branch analysis. In the final product the pipeline consists of building, unit testing and SonarQube analysis of the project. In the future there is a need to find a suitable tool for code coverage that works with both frameworks.

Keywords: Microsoft Azure, Azure DevOps, Azure Pipelines, Continuous Integration, SonarQube, Software quality, Code quality

Sisällys

Lyhenteet ja käsitteet

1	Johdanto	1
2	Ohjelmiston laatu	2
2.1	Laadun osa-alueet	2
2.2	Koodin laatu	7
2.3	Laadunhallinta	9
2.3.1	Testaus	10
2.3.2	SonarQube	13
2.3.3	WhiteSource (Mend)	14
3	Microsoft Azure	15
3.1	Pilvipalveluista yleisesti	15
3.2	Azure DevOps	18
3.2.1	Azure Pipelines	22
4	Pull request -validointiputki	27
4.1	Projektin tiedot	28
4.2	Tiimin haastattelu	28
4.3	WhiteSource (Mend)	30
4.4	SonarQube	31
4.5	Testikattavuus	32
4.6	Valmis putki ja sen käyttö	33
4.7	Jatkokehitysideoita	42
5	Yhteenveto	43
	Lähteet	44

Lyhenteet ja käsitteet

- ADO: *Azure DevOps*. Microsoft Azuren tarjoama palvelu sovellusten kehitykseen ja julkaisuun.
- IDE: *Integrated Development Environment*. Integroitu kehitysympäristö, joka tarjoaa monia apuvälineitä ja työkaluja ohjelmointiprojektin toteutukseen. IDE sisältää usein tekstieditorin, debuggauksen eli työkalun, jonka avulla voi etsiä ja selvittää ohjelmointivirheitä, automaattisen koonnin sekä monia muita. Usein IDE on ohjelmointikielikohtainen ja tarjoaa kyseiselle kielelle yksilöityjä työkaluja, kuten refaktoroinnin, virheiden havaitsemisen, koodin automaattisen täydennyksen sekä joitain valmiita perusmetodeja. IDE voi myös sisältää muita hyödyllisiä työkaluja, kuten terminaalin tai versionhallinnan.
- a: *Continuous Integration (CI)*. Menetelmä, jota sovelluskehitystiimit käyttävät lähdekoodimuutosten yhdistämiseen yhteen ohjelmistotuotantoprojektiin. Jatkuva integraatio auttaa ylläpitämään laadukasta koodia ja löytämään ohjelmointivirheet helpommin ja varhaisessa vaiheessa, sillä virhe löytyy todennäköisesti viimeksi lisätystä koodista. Näin ollen myös muutosten palautus edelliseen tilaan aiheuttaa vähemmän häiriötä, kun muutokset on integroitu pienissä osissa. Varhaisessa vaiheessa toteutettu automaattinen testaus lisää koodin laatua. [1.]
- Ohjelmistokehys: *Framework*. Nimensä mukaisesti kehys, joka tarjoaa ge-
neeriset, uudelleenkäytettävät toiminnallisuudet, joiden sisältö määritellään omalla koodilla, jota kehys kutsuu. Ohjelmistokehysten tarkoitus on nopeuttaa sovelluskehitystä, kun kaikkea ei tarvitse kirjoittaa alusta asti uudelleen.
- Ohjelmointirajapinta: *API, Application Program Interface*. Määritelmä, jonka avulla sovellukset ja palvelut voivat kommunikoida keskenään.

PR: *Pull Request*. Käytetään myös nimitystä *Merge Request* tietovarastopalvelusta riippuen. Jatkuvan integraation keskeinen toiminto, jonka avulla kehittäjä kertoo projektin muille kehittäjille tekemistään muutoksista ennen niiden yhdistämistä yhteiseen päähaaraan.

Projekti: Tämän opinnäytetyön kohteena oleva sovellus ei ole julkinen referenssi, joten siihen viitataan tässä työssä *projektilla*.

Päähaara: *Master branch*. Kehitystiimin yhteinen kehityshaara, johon kehittäjät yhdistävät tekemänsä muutokset. Myös *main branch* on nimitys päähaaralle, ja monet palveluntarjoajat ovat siirtyneet Git Projectin mukana käyttämään päähaarasta nimitystä *main masterin* sijaan. *Master* on koettu loukkaavaksi sen historiallisen yhteyden vuoksi, jonka vuoksi Git mahdollisti päähaaran nimen manuaalisen määrittelyn vuonna 2020 inkluusion lisäämiseksi. Myös Azure Repos lähti mukaan nimeämiskäytännön muutokseen, ja uusien projektien päähaaran nimi on oletuksena *main*. Tässä opinnäytetyössä on kyseessä kuitenkin vanha projekti, jonka päähaarasta on käytössä englanninkielinen nimi *master*. [2.; 3.]

Tietovarasto: *Repository*. Paikka, jossa sovelluksen lähdekoodi sijaitsee. Tässä opinnäytetyössä käsiteltävän projektin lähdekoodi sijaitsee *Azure Repos* -palvelussa käyttäen *Git*ä versionhallinnassa.

1 Johdanto

Tämän insinööriyön toimeksiantaja Oy Samlink Ab panostaa tällä hetkellä pilvipalveluihin siirtymiseen ja niiden kehittämiseen. Samlink on suomalainen ohjelmistoyritys, jonka asiakkaat ovat pääasiassa finanssialan toimijoita, kuten sijoituspalveluyrityksiä, luottolaitoksia ja pankkeja. Sen tunnetuimpia asiakkaita ovat muiden muassa POP-pankki, Säästöpankki ja Oma Säästöpankki. Samlinkin emoyhtiö on Kyndryl, jonka alaisuuteen se siirtyi alkuvuodesta 2022.

Samlinkin Microsoft technologies -tiimin vastuulla on erään rahoitustenhallintajärjestelmän kehitys. Sovelluksen kehitys ja julkaisu tapahtuu Microsoftin Azure-pilvipalvelussa: julkaisu tapahtuu Azuren Web Apps -alustalla sekä jatkuva integraatio ja toimittaminen Azure DevOps -alustalla. Projekti on läpikäymässä uudistusta, jonka tavoitteena oli lisätä automatisaatiota ja sen avulla projektin laatua. Pääsin mukaan tähän uudistukseen kehittämään jatkuvan integraation ratkaisuja Azure DevOpsia hyödyntäen, minkä tavoitteena on parantaa koodin laatua ja vähentää häiriöitä päähaarassa ja käyttöönotoissa.

Lähtötilannetta selvitin kehitystiimille lähetettävällä anonyymillä kyselyllä, jonka tavoitteena on tuoda julki parannuskohteita ja auttaa luomaan kehittäjiä helpottavia automaatioita, joiden avulla koodin laatua ja eheyttä on helpompi ylläpitää. Projektin lähtötilanteessa *Pull Request*:in hyväksymisen vaatimuksena on kahden ihmisten suorittama koodikatselmointi. Pull Requestilla tarkoitetaan toimintatapaa, jolla kehittäjä kertoo muulle kehitystiimille tekemistään koodimuutoksista ennen niiden lisäystä tiimin yhteiseen päähaaraan. Testaus ja muut laatuanalyysit suoritettiin vasta päähaarassa, minkä seurauksena ongelmat havaittiin usein vasta uuden koodin päädyttyä päähaaraan, ja tämä aiheutti viivästyksiä kehitystyöhön. Myöskään testikattavuutta tai koodin laatua ei seurattu, joten projektiin oli päässyt kertymään jonkin verran teknistä velkaa.

Tavoitteenani oli luoda kehitystiimin työtä helpottava jatkuvan integraation putki, joka keventäisi kehittäjien vastuulla olevaa laadun valvontaa automatisaation avulla. Luomalla PR-validointiputken, joka suorittaisi koonnin,

yksikkötestauksen sekä lisäksi laatua analysoivia työkaluja tehostettaisiin tiimin työskentelyä automatisoimalla aikaa vieviä koodin laadun tarkistuksia.

2 Ohjelmiston laatu

2.1 Laadun osa-alueet

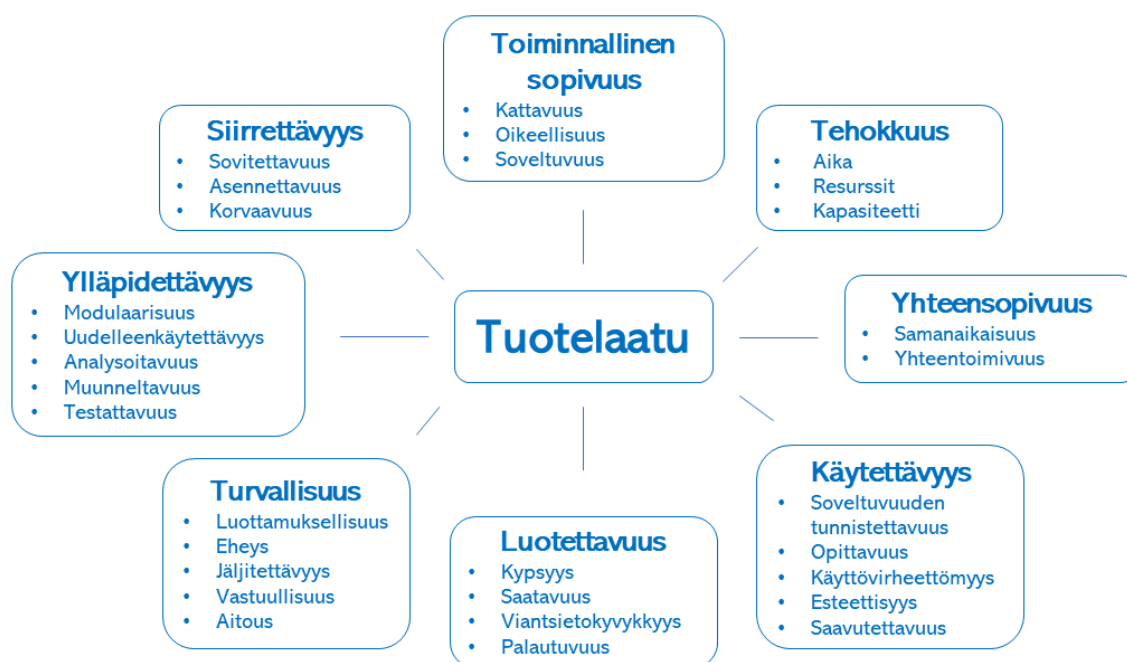
Ohjelmiston laadulla tarkoitetaan sitä astetta, jolla ohjelmisto täyttää sille asetetut odotukset ja vaatimukset. Laatua voidaan arvioida erilaisten mittareiden avulla, kuten luotettavuudella, käytettävyydellä, suorituskyvillä ja ylläpidettävyydellä. Laadukas ohjelmisto ei sisällä ohjelmointivirheitä, on miellyttävä ja helppo käyttää, suoriutuu tehtävistä nopeasti ja tehokkaasti ja kestää muutoksia, kuten uusien toiminnallisuuksien lisäämisen. [4.]

Ohjelmiston laadulle on olemassa kansainvälisiä standardeja. ISO, eli *International Organization for Standardization*, on riippumaton kansainvälinen järjestö, joka luo asiantuntijoiden avulla kansainvälisiä standardeja eri aloille. Tietoteknisten järjestelmien ja ohjelmistojen laadulle on olemassa oma standardiperhe ISO/IEC 25000. ISO/IEC 25010, *Systems and software Quality Requirements and Evaluation* eli SQuaRE, arvioi järjestelmän ja ohjelmistojen laatua sekä itse tuotteen että sen käytön kannalta. Sen mukaan tuotteen laatua mitataan kuvassa 1 kuvattujen seuraavien kahdeksan laatupiirteen ja niiden alle kuuluvien laatuominaisuuksien avulla: [5.]

- **Toiminnallinen sopivuus:** Taso, jolla tuote tuottaa vaaditut toiminnallisuudet ennalta määritellyssä käyttötilanteessa.
 - (a) **Kattavuus:** Taso, jolla tuotteen toiminnot kattavat määritellyt tehtävät ja käyttötavoitteet.
 - (b) **Oikeellisuus:** Taso, jolla tuote tuottaa oikeat tulokset määritellyllä tarkkuudella.
 - (c) **Soveltuvuus:** Taso, jolla tuote helpottaa määriteltyjen tehtävien ja tavoitteiden saavuttamista.
- **Tehokkuus:** Suorituskyvyn taso resursseihin suhteutettuna ennalta määritellyssä käyttötilanteessa.

- (a) **Aika:** Taso, jolla tuote saavuttaa asetetut vaste- ja suoritusajat määritellyjä tehtäviä suorittaessa.
 - (b) **Resurssit:** Taso, jolla tuotteen käyttämät resurssit tehtäviä suorittaessa vastaavat määrittelyjä.
 - (c) **Kapasiteetti:** Taso, jolla tuotteen enimmäiskapasiteetti, kuten käyttäjien tai tallennettavien tiedostojen enimmäismäärä vastaa määrittelyä.
- **Yhteensopivuus:** Taso, jolla tuote pystyy vaihtamaan tietoa ja/tai jakamaan yhteisen laitteiston tai ohjelmistoympäristön muiden tuotteiden kanssa.
 - (a) **Samanaikaisuus:** Taso, jolla tuote pystyy suoriutumaan siltä vaadituista tehtävistä tehokkaasti vaikuttamatta muiden saman ympäristön ja resurssien jakavien tuotteiden toimintaan.
 - (b) **Yhteentoimivuus:** Taso, jolla tuote pystyy vaihtamaan tietoa muiden tuotteiden kanssa.
- **Käytettävyys:** Taso, jolla tuotteen määritellyt käyttäjät voivat suorittaa ennalta määritellyistä tehtävistä tehokkaasti, vaikuttavasti ja tyydyttävästi.
 - (a) **Soveltuvuuden tunnistettavuus:** Taso, jolla tuotteen käyttäjät voivat tunnistaa sovelluksen sopivan heidän käyttötarpeisiinsa.
 - (b) **Opittavuus:** Taso, jolla määritellyjen käyttäjien on mahdollista oppia käyttämään tuotetta määritellyissä käyttötilanteissa tehokkaasti, riskittä ja tyydyttävästi.
 - (c) **Helppokäyttöisyys:** Taso, jolla tuotteen käyttöä ja hallinnointia helpottavat ominaisuudet ovat.
 - (d) **Käyttövirheettömyys:** Taso, jolla tuote ennalta ehkäisee käyttäjien tekemiä virheitä.
 - (e) **Esteettisyys:** Taso, jolla tuotteen käyttöliittymä mahdollistaa miellyttävän ja tyydyttävän käyttökokemuksen.
 - (f) **Saavutettavuus:** Taso, jolla mahdollisimman moni käyttäjä erilaisista henkilökohtaisista rajoitteista huolimatta pystyy suorittamaan määritellyt käyttötapausten määritellyissä tilanteissa.
- **Luotettavuus:** Taso, jolla tuote suoriutuu sille määritellyistä tehtävistä tietyissä olosuhteissa ennalta määritellyn ajan.
 - (a) **Kypsyys:** Taso, jolla tuote vastaa normaalissa käytössä luotettavuuden tarpeisiin.
 - (b) **Saatavuus:** Taso, jolla tuote on toiminnassa ja käytettävissä tarpeen vaatiessa.
 - (c) **Viansietokyvykkyys:** Taso, jolla tuote toimii tarkoitetulla tavalla laitteisto- tai ohjelmistovioista huolimatta.
 - (d) **Palautuvuus:** Taso, jolla tuote kykenee virheen tai häiriön satuessa palauttamaan toiminnallisuuden ja datan.

- **Turvallisuus:** Taso, jolla tuote suojaa tietoja ja dataa niin, että käyttäjillä tai muilla tuotteilla on niiden valtuuksia vastaava pääsy tietoihin ja dataan.
 - (a) **Luottamuksellisuus:** Taso, jolla tuote pitää huolen siitä, että dataan on pääsy vain siihen valtuutetuilla henkilöillä.
 - (b) **Eheys:** Taso, jolla tuote estää luvattoman pääsyn sekä datan tai muun sovelluksen osan luvattoman käsittelyn.
 - (c) **Jäljitettävyy:** Taso, jolla tekojen ja tapahtumien voidaan todistaa tapahtuneen ilman, että niitä voi jälkikäteen kieltää tapahtuneen.
 - (d) **Vastuullisuus:** Taso, jolla tuotteen käyttö on jäljitettävissä takaisin käyttäjänsä.
 - (e) **Aitous:** Taso, jolla kohteen tai resurssin identiteetti voidaan todentaa väitetyksi.
- **Ylläpidettävyy:** Taso, jolla tuotteen ylläpitäjät voivat tehokkaasti ja vaikuttavasti muunnella tuotetta.
 - (a) **Modulaarisuus:** Taso, jolla tuote on luotu itsenäisistä osista, joiden muutoksilla on mahdollisimman pieni vaikutus tuotteen muihin osiin.
 - (b) **Uudelleenkäytettävyy:** Taso, jolla tuotetta voi käyttää uudelleen tai osana uusia tuotteita.
 - (c) **Analysoitavuus:** Taso, jolla tuotteeseen tehtyjen muutosten vaikutuksia on mahdollista arvioida, havaita puutteita tai virheidensä syitä tai tunnistaa paranneltavia osia.
 - (d) **Muunneltavuus:** Taso, jolla tuote on muunneltavissa tehokkaasti aiheuttamatta ongelmia muissa osissa tuotetta tai heikentämättä jo olemassa olevaa laatua.
 - (e) **Testattavuus:** Taso, jolla tuotteelle voidaan määritellä testikriteerit, suorittaa kyseiset testit ja arvioida testien tulosta.
- **Siirrettävyy:** Taso, jolla tuotetta voidaan tehokkaasti siirrellä järjestelmästä, laitteistosta tai muusta käyttöympäristöstä toiseen.
 - (a) **Sovitettavuus:** Taso, jolla tuote on sovitettavissa eri käyttö- ja toimintaympäristöihin sekä tuotteen skaalautuvuus.
 - (b) **Asennettavuus:** Taso, jolla tuote on asennettavissa ja poistettavissa määritellyissä käyttöympäristöissä.
 - (c) **Korvaavuus:** Taso, jolla tuote voi korvata jonkin toisen, vastaavaan käyttöön tarkoitetun tuotteen.

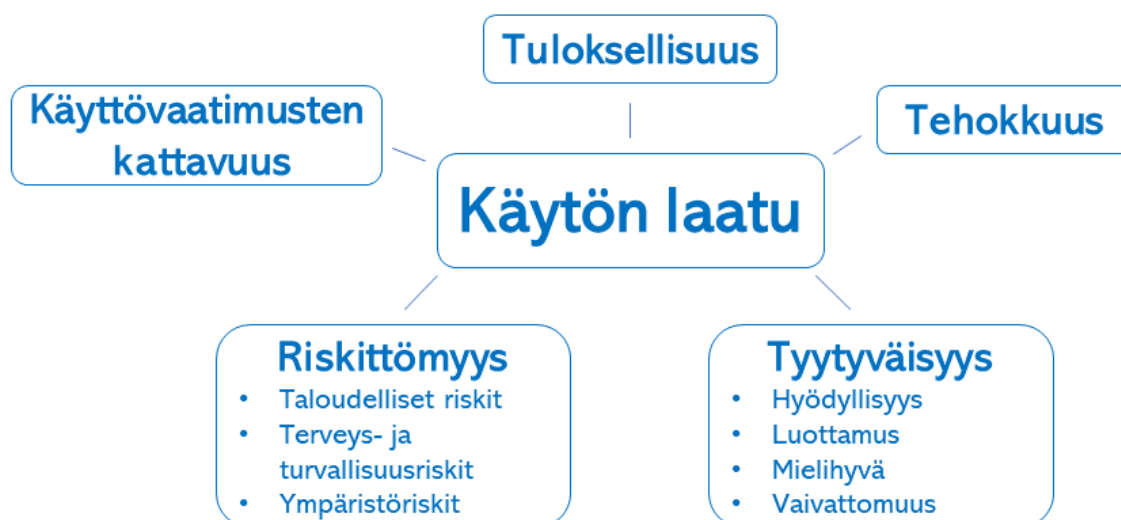


Kuva 1. Ohjelmiston tai järjestelmän tuotelaadun laatupiirteet ja laatuominaisuudet.

Käytön laatua mitataan kuvassa 2 kuvattujen seuraavan viiden laatupiirteen ja niiden alakategorioiden mukaan: [5.]

- **Tuloksellisuus:** Kuinka täsmällisesti ja täydellisesti käyttäjä saavuttaa vaaditun tehtävän.
- **Tehokkuus:** Kuinka paljon resursseja (esimerkiksi aika, materiaalit, käytön kustannus) järjestelmän tai ohjelmiston käyttö vaatii suhteessa käyttäjän saavuttaman tuloksen täsmällisyyteen ja tarkkuuteen.
- **Tyytyväisyys:** Kuinka tyytyväinen käyttäjä on määritellyn käyttötapausten suorittamisen jälkeen.
 - (a) Hyödyllisyys: Kuinka tyytyväinen käyttäjä on järjestelmän tai ohjelmiston käytöstä saatuun hyötyyn.
 - (b) Luottamus: Kuinka hyvin käyttäjä tai muu sidosryhmän jäsen voi luottaa siihen, että järjestelmä tai ohjelmisto käyttäytyy odotetulla tavalla.
 - (c) Mielihyvä: Kuinka paljon mielihyvää saadun tuloksen saavuttaminen tuottaa käyttäjälle.
 - (d) Vaivattomuus: Kuinka tyytyväinen käyttäjä on fyysiseen mukavuuteen.

- **Riskittömyys:** Kuinka järjestelmä tai ohjelmisto estää ja hallitsee riskejä suhteessa niiden potentiaaliseen vaikutukseen.
 - (a) Taloudelliset riskit: Kuinka hyvin järjestelmä tai ohjelmisto minimoi riskejä taloudelliselle tilanteelle, tehokkaalle toiminnalle maineelle, kaupalliselle omaisuudelle tai muille resursseille kyseisessä käyttötilanteessa.
 - (b) Terveys- ja turvallisuusriskit: Kuinka hyvin järjestelmä tai ohjelmisto minimoi ihmisiin kohdistuvia riskejä kyseisessä käyttötilanteessa.
 - (c) Ympäristöriskit: Kuinka hyvin järjestelmä tai ohjelmisto minimoi omaisuuteen tai ympäristöön kohdistuvia riskejä kyseisessä käyttötilanteessa.
- **Käyttövaatimusten kattavuus:** Kuinka hyvin järjestelmä tai ohjelmisto toteuttaa yllä mainittuja neljää laatupiirrettä määritellyissä käytötapauksissa sekä niiden ulkopuolella.



Kuva 2. Ohjelmiston tai järjestelmän käytön laadun laatupiirteet ja laatuominaisuudet.

Laadukas ohjelmistotuote on elintärkeä yrityksen tuloksen ja maineen kannalta. Koska sertifikaatit ovat hyvin yksityiskohtaisia, voi sen hankkiminen tulla pienille yrityksille kalliiksi. Standardit sisältävät kuitenkin yleisesti hyväksytyjä, hyviä alan käytäntöjä ja mittareita, joita voi ottaa osaksi omia prosesseja ja näin ylläpitää laatua. [4.]

2.2 Koodin laatu

Koodin ei tarvitse olla ensimmäisellä kerralla täydellistä. Koodin refaktorointi on tärkeä osa ohjelmointia. Refaktoroinnilla tarkoitetaan olemassa olevan koodin muotoilua uudelleen ilman, että sen ulkoinen käyttäytyminen muuttuu. Refaktoroinnin popularisoi Martin Fowler, englantilainen ohjelmistoinsinööri kirjassaan *Refactoring: Improving the Design of Existing Code* (1999) [6]. Perinteisesti ajateltuna ohjelmistokehitys nojaa ensisijaisesti hyvään etukäteen tehtyyn suunnitteluun ja itse ohjelmointi on toissijaista. Ajan myötä ohjelmistoa kuitenkin kehitetään edelleen, lisätään uusia toiminnallisuuksia ja poistetaan tarpeettomia, jolloin alkuperäisen suunnitelman mukainen rakenne vähitellen häviää ja uuden koodiin lisäämisestä tulee yhä hitaampaa ja haastavampaa. Refaktoroinnin lähtökohta on päinvastainen: etukäteen tehdyn huolellisen suunnittelun ja rakenteen määrittelyn sijaan keskistytään luomaan rakenteita ja malleja sitä mukaa kun, koodia kirjoitetaan. Lopputuloksena on ohjelmisto, jonka rakenne pysyy hyvänä kehityksen myötä.

Refaktoroinnin tarpeellisuuden arviointi on subjektiivista ja ohjelmointikielikohdista. *Refactoring*-kirjaa oli mukana kirjoittamassa myös muiden muassa ohjelmistoinsinööri Kent Beck, jonka keksimä termi *Code Smell* otettiin laajemmin yleiseen käyttöön kirjan saaman suosion myötä. Code smellillä tarkoitetaan tunto-merkkejä, jotka vihjaavat refaktoroinnin olevan tarpeellista. Ne eivät ole varsinaisesti ohjelmointivirheitä, mutta ovat käytäntöjä ja rakenteita, jotka tekevät koodista vaikeasti ymmärrettävää, ylläpidettävää ja muunneltavaa. Yksi yleisin code smell on esimerkiksi *duplicate code* eli toistuva koodi, jossa sama koodi tai logiikka toistuu useammassa kohtaa lähdekoodia. Toistuvaa koodia syntyy usein, kun tiettyä toiminnallisuutta tarvitaan useammassa kohdassa lähdekoodia ja se kopioidaan sellaisenaan esimerkiksi luokalta toiselle sen sijaan, että se abstrahoitaisiin, eli tehtäisiin yleiskäyttöinen toiminnallisuus, josta puuttuu yksityiskohdat, erilliseksi, uudelleen käytettäväksi funktioksi tai moduuliksi.

Clean code eli siisti koodi -termin teki tunnetuksi amerikkalainen ohjelmistoinsinööri Robert C. Martin kirjassaan *Clean Code: A Handbook of Agile Software*

Craftmanship (2008) [7], josta on tullut yksi aiheen merkkiteoksista. Siistillä koodilla tarkoitetaan koodia, joka on hyvin jäsenneltyä, helppoa ymmärtää ja ylläpitää. Siisti koodi seuraa ohjelmistokehityksen parhaita käytäntöjä, suunnittelumalleja ja standardeja. Näiden avulla tavoitteena on saada aikaan koodia, jonka jokainen tiimin jäsen ymmärtää ja joka on ihmiselle helposti luettavaa hyvien luokkien, metodien ja muuttujien nimeämiskäytäntöjen ja turhan kompleksisuuden poiston myötä, helppoa ylläpitää komponenttien, luokkien ja metodien löyhän kytkennän, uudelleenkäytettävyyden ja pienen koon ansiosta, jolloin toiminnallisuuksien poisto, päivitys ja uusien lisäys on helppoa, ja näiden toteutuksella ei ole vaikutusta kyseisen osan laajuuden ulkopuolelle.

Siistillä koodilla on vaikutusta myös muuhunkin kuin vain koodin luettavuuteen ja ymmärrettävyyteen. Suurin osa ohjelmointivirheistä syntyy, kun muutetaan jo olemassa olevaa koodia. Tämä on seurausta huonosti ymmärrettävästä koodista, kun kehittäjän on vaikeaa hahmottaa, mihin kaikkeen tehdyt muutokset tulevat vaikuttamaan. Siisti koodi ei takaa, että ohjelmointivirheitä ei koskaan synny, mutta se auttaa vähentämään sovelluksen virheiden vakavuutta ja esiintymistä: Kun koodi on helposti ymmärrettävää ja luettavaa, eikä se sisällä turhan monimutkaista logiikkaa, ovat ohjelmointivirheetkin helpompia havaita. Löyhästi toisiinsa kytkettyjen ohjelmiston osien muuttaminen ja lisääminen vaikuttaa ainoastaan kyseiseen osaan, eikä aiheuta ketjureaktiota, joka vaatisi muutoksia kumuloituvasti muissa osissa ohjelmisto. Koodin testattavuus paranee myös, mitä vähemmän sillä on riippuvuuksia. Automaattisen testauksen avulla ohjelmointivirheiden havaitseminen helpottuu ja nopeutuu. Näin vältetään virheiden päätyemiseltä tuotantoon.

Vaikka siistin koodin saavuttamiseen voi kulua resursseja, kuten aikaa ja sen myötä rahaa. Pitkällä aikavälillä se on hyvä sijoitus, sillä nämä resurssit säästävät myöhemmin ylläpidon ja jatkokehityksen aikana. On kuitenkin hyvä ottaa huomioon refaktoroinnin tarvetta arvioitaessa, mikä on kyseisen ohjelmiston osan elinkaari. Jos osa on esimerkiksi tilapäinen ja korvataan lähitulevaisuudessa, ei sen refaktorointiin ole välttämättä järkevää käyttää resursseja. [8.]

2.3 Laadunhallinta

Jotta koodin laatu ei jäisi pelkästään ohjelmistokehittäjien subjektiivisen tarkastelun varaan, on sen tarkasteluun olemassa automatisoituja työkaluja. Näiden tarkoituksena on helpottaa alan hyvien käytäntöjen ja standardien noudattamista.

Nykyään monille ohjelmointikielille on olemassa IDE, eli integroitu kehitysympäristö (*Integrated Development Environment*), joka tarjoaa runsaasti sisäänrakennettuja refaktorointia automatisoivia toimintoja. Esimerkiksi muuttujan uudelleennimeäminen nimeää uudelleen kaikki muuttujan esiintymät, eikä kehittäjän tarvitse käydä koko lähdekoodia läpi etsien jokaista muuttujan esiintymää. Lisäksi useisiin IDE:in saa asennettua kattavia työkaluja helpottamaan koodin laadun seuraamista. Ne antavat esimerkiksi parannusehdotuksia, kertovat syyn korjaamisen tarpeelle sekä antavat esimerkkejä reaaliajassa samalla, kun koodia kirjoitetaan, jolloin ongelmakohtien havaitseminen ja korjaaminen helpottuu. Tällaisia IDE:n laajennuksia ovat esimerkiksi SonarLint ja ESLint.

CISQ eli *Constortium for Information & Software Quality* on yhdysvaltalainen voittoa tavoittelematon, avoimen lähdekoodin järjestö, joka pyrkii luomaan ohjelmistoalan standardeja ja tapoja mitata ohjelmistojen laatua alan ammattilaisten avulla. CISQ:n luoman teknisen velan standardin avulla on mahdollista mitata, kuinka suurta osaa lähdekoodista tekninen velka koskee, yksittäisten teknisen velan standardin rikkovien tapausten määrän sekä kustannuksen näiden korjaamiseksi. [9.] Tekninen velka on yhdysvaltalaisen ohjelmoijan Howard G. Cunninghamin keksimä metafora, jolla tarkoitetaan sitä ylimääräistä vaivaa, jonka uuden ominaisuuden lisääminen ohjelmistoon aiheuttaa, kun koodi ei ole siistiä ja laadukasta. [8.]

Viranomaiset ja kriittiset alat, kuten turvallisuus-, finanssi- ja terveydenhuoltoalat, voivat vaatia sovelluksen laadun todistamista. Järjestelmän tai ohjelmistotuotteen laadun ISO/IEC 25010-standardin lisäksi on olemassa lähdekoodin laadulle oma standardi ISO/IEC 5055:2021, johon jatkossa viitataan tässä

opinnäytetyössä lyhyemmin ISO 5055. Se otettiin käyttöön vuonna 2021 ja se mittaa sovelluksen neljää seuraavaa liiketoiminnalle kriittistä tekijää: turvallisuus, luotettavuus, suorituskyvyn tehokkuus ja ylläpidettävyys. Näiden avulla voidaan päätellä, kuinka luotettava ja kestävä sovellus on. Se on ensimmäinen kansainvälinen standardi, joka analysoi sovelluksen lähdekoodia havaitakseen arkkitehtuurillisia virheitä ja haitallisia koodauskäytäntöjä, jotka voivat aiheuttaa vakavia riskejä ja mittavia kustannuksia edellä mainittuihin neljään tekijään. ISO 5055 kattaa neljä kahdeksasta ISO 25010 -standardista, joka on laajempi kokonaisuus systeemien ja sovellusten laadulle. ISO-sertifioidulla sovelluksella voi olla merkittävä markkinaetu, sillä kansainvälisesti hyväksytty, riippumaton sertifikaatti voi avata ovia kansainvälisiltä markkinoilta ja tarjota asiakkaalle vahvistuksen laadukkaasta sovelluksesta ja näin vaikuttaa päätökseen siitä, kenen tuottajan sovellukseen päädytään. Koska ISO 5055 on varsin uusi standardi, tällä hetkellä ainoa ISO 5055 -standardin täyttävä ja sertifiointin tarjoava sovel-lusanalyysityökalu on CAST Imaging. [10; 11.]

2.3.1 Testaus

Testaus on olennainen osa ohjelmiston laadun varmistamista. Testauksen tavoitteena on varmistua, että ohjelmisto toimii toivotulla tavalla ja vähentää virheiden mahdollisuutta. Se voidaan jakaa kahteen pääkategoriaan: funktionaaliseen testaukseen (*Functional testing*) ja ei-funktionaaliseen testaukseen (*Non-functional testing*, eli NFT) Funktionaalaisella testauksella selvitetään, että ohjelmisto ja sen osat toimivat ennalta määritellyllä tavalla. Ei-funktionaalinen testaus keskittyy funktionaalisen testauksen ulkopuolelle jääviin seikkoihin, kuten käytettävyyteen, suorituskykyyn ja skaalautuvuuteen. [12.] Vaikka erilaisia testitapauksia on olemassa runsaasti, on täysin kattava testaus käytännössä mahdotonta lähes loputtomien muuttujayhdistelmien vuoksi. Siksi testaukseen on syytä valita testitapaukset harkiten niin, että ne kattavat mahdollisimman laajasti eri asioita. Jokainen lause, ehto, toistorakenne ja polku suoritetaan alusta loppuun vähintään kerran testauksen aikana. Silmukoiden kohdalla erilaisia polkuja voi kuitenkin olla ääretön määrä, jolloin on hyvä testata ainakin yleisimmät polut sekä muutama tarkoin valittu harvinaisempi polku. [13.]



Kuva 3. Esimerkkejä funktionaaliseen ja ei-funktionaaliseen testaukseen kuuluvista testeistä.

Testausta voidaan tehdä monelta eri näkökulmalta, ja niiden käyttö vaihtelee tilannekohtaisesti. Funktionaalisiin testeihin kuuluvalla yksikkötestauksella varmistetaan, että luokat, komponentit ja moduulit, tai niiden funktiot ja metodit toimivat halutulla tavalla muista osista eristettyinä. Yksikkötestaus on kehittäjien vastuulla, ja usein yksikkötestit kirjoitetaan ennen itse koodin kirjoittamista, jolloin puhutaan testivetoisesta kehityksestä eli TDD:stä (*Test Driven Development*). Yksikkötesteillä on mahdollista havaita ohjelmointivirheet varhaisessa vaiheessa, jolloin niiden korjaaminen vie vähemmän aikaa, kun testaus suoritetaan säännöllisesti. [14; 15.]

Integraatiotestauksen tehtävä on varmistaa, että sovelluksen eri komponentit toimivat yhdessä ja välittävät dataa toisilleen odotetulla tavalla. Integraatiotestaus suoritetaan yleensä yksikkötestauksen jälkeen, kun on varmistettu, että sovelluksen eri osat toimivat eristettyinä muista. Integraatiotestauksesta voi vastata kehittäjä itse tai sovellustestaaja. [14; 15.]

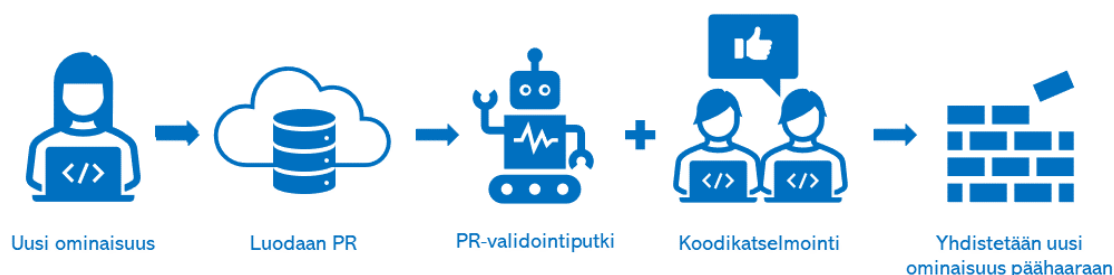
NFT-testeihin kuuluvalla suorituskykytestauksella selvitetään, kuinka sovellus suoriutuu erilaisten kuormitusten alla. Sen avulla voidaan mitata, kuinka luotettava, nopea ja skaalautuva sovellus on. Erilaisia suorituskykytestejä ovat esimerkiksi kuormitustestit, joissa simuloidaan esimerkiksi suurta kävijämäärää

verkkosivulla, ja näin selvitetään, kuinka hyvin järjestelmä selviytyy tosielämän tilanteista. Kuormitustestien avulla voidaan havaita hitaita vasteaikoja tai lisääntyntä resurssien käyttöä. Kestävyytestauksessa selvitetään, kuinka järjestelmä käyttäytyy ajan kuluessa ja auttaa havaitsemaan esimerkiksi muistivuoja ja muita suorituskyvyn alenemia. Skaalautuvuustestien avulla voidaan havaita suorituskyvyn pullonkauloja ja muita ongelmia, kun arvioidaan järjestelmän kykyä skaalata resursseja vastaamaan kasvavaa kysyntää, kuten suuria käyttäjämääriä tai lisääntyntä datan käsittelytarvetta. Stressitestisteissä selvitetään järjestelmän käyttäytymistä ja kestävyyttä sille asetettujen kestävyysodotusten ulkopuolella. Suorituskvykyteteille on olemassa työkaluja ja testauskehyskiä, kuten Apache JMeter ja Gatling, jotka automatisoivat testauksen ja tarjoavat yksityiskohtaisia mittauktuloksia. [14; 15.]

Lisäksi testauksessa on hyvä ottaa huomioon myös käytettävyyden testaus sekä lopuksi yleensä asiakkaan tai loppukäyttäjän tekemä hyväksymistestaus, jolla varmistetaan, että sovellus toimii kokonaisuutena sen käyttötarkoituksen mukaisesti ja on valmis käyttöönottoon. [14; 15.]

End-To-End-testaus (E2E-testaus) käy läpi koko sovelluksen kaikki kerrokset, riippuvuudet ja käytön vaiheet alusta loppuun käyttöliittymästä aina tietokantaan asti, ja testaa niiden toiminnallisuuden ja suorituskvyn tuotantoa vastaavissa olosuhteissa. Se simuloi oikeita käyttötapauksia ja hyödyntää testidataa jäljentääkseen tosielämän tilanteita. E2E-testaus ei kuitenkaan korvaa muita testaus-tapoja, vaan tarjoaa täydennystä muille kattaville testeille. [16.]

Testaus on mahdollista automatisoida ja ottaa osaksi jatkuvan integraation ratkaisuja. Jatkuvalla integraatiolla tarkoitetaan kehitystiimin tekemien lähdekoodi-muutosten säännöllistä, jopa useita kertoja päivässä tapahtuvaa yhdistämistä yhteiseen ohjelmistoprojektiin. Tätä lähdekoodiin yhdistämispyyntöä kutsutaan *Pull Request*iksi, johon viitataan tässä opinnäytetyössä jatkossa lyhenteellä PR.



Kuva 4. Kehittäjä luo uutta ominaisuutta omassa kehityshaarassa. Haaraa säilytetään ulkoisessa tietovarastossa, tämän opinnäytetyön kohdalla Azure Repos -palvelussa. Kun ominaisuus on valmis, kehittäjä luo siitä *Pull Requestin*, eli pyynnön lisätä uusi ominaisuus osaksi kehitystiimin yhteistä päähaaraa. Jotta PR voidaan hyväksyä, on sen läpäistävä validointiputki, joka suorittaa koonnin, yksikkötestit ja laatuanalyysit automaattisesti. Tämän lisäksi kahden muun kehitystiimiläisen on hyväksyttävä ehdotetut muutokset. Kun validointiputki ja koodikatselmointi on hyväksytty, yhdistetään uusi ominaisuus päähaaraan osaksi yhteistä projektia.

PR:n yhteydessä on mahdollista ajaa testit automaattisesti, jolloin ongelmat voidaan havaita ennen kuin ne päätyvät yhteiseen projektiin aiheuttaen ongelmia myös muille kehittäjille. Testien ajo saattaa viedä runsaasti aikaa, joten on syytä pohtia, mitä testejä kannattaa ajaa PR:n yhteydessä. Kaikkia eri testityyppejä ei välttämättä ole ajankäytön kannalta järkevää suorittaa tässä vaiheessa, mutta vähintään yksikkötestaus on syytä ajaa säännöllisesti uuden koodin osalta ennen yhdistämistä.

2.3.2 SonarQube

SonarQube on SonarSourcen kehittämä avoimen lähdekoodin ohjelmisto koodin laadunvalvontaan. SonarQube automatisoi ohjelmointivirheiden, turvallisuusriskien ja code smellien havaitsemiseen, minkä avulla kehittäjät voivat ylläpitää tervettä koodipohjaa saamallaan palautteella kehitystyön varhaisessa vaiheessa. Tämä auttaa ylläpitämään muun muassa turvallista, luettavaa ja ylläpidettävää koodipohjaa. SonarQube muun muassa analysoi koodin kompleksisuutta, eli laskee polkujen määrän koodin alusta loppuun. Esimerkiksi ehtolauseet ja silmukat lisäävät näitä polkuja. Lisäksi se etsii lähdekoodista ohjelmointi- ja syntaksivirheitä, yleisten ohjelmointistandardien vastaisia käytäntöjä, kuten

code smellejä, toistuvaa koodia tiedosto- ja rivitasolla, tietoturva-aukkoja sekä muita koodin laatuun ja turvallisuuteen vaikuttavia tekijöitä, kuten teknisen velan määrää. [17.] SonarQube tukee yli kolmeakymmentä eri ohjelmointikieltä ja -kehystä, ja se on mahdollista integroida suosituimpiin DevOps-alustoihin. SonarQuben suorittaman analyysin pohjalta on mahdollista määrittää sääntö PR:n yhteyteen, joka estää koodin yhdistämisen päähaaraan, mikäli ennalta määritellyt laatuvaatimukset eivät toteudu uuden koodin osalta. Näin voidaan entisestään varmistua siitä, että sovelluksen lähdekoodi pysyy turvallisena ja ylläpidettävänä koko kehitystiimille. [18.]

2.3.3 WhiteSource (Mend)

Haavoittuvien ja vanhentuneiden komponenttien aiheuttamat tietoturvariskit ovat olleet nousussa viime vuosien aikana. Tämä on havaittu niin CISQ:n vuoden 2022 *Cost of poor software quality in the U.S.* -raportissa [19] että The OWASP Foundation:in vuoden 2021 *Top 10 Web Application Security Risks* -listalla sijalla kuusi [20]. OWASP eli The Open Worldwide Application Security Project on voittoa tavoittelematon järjestö, joka pyrkii parantamaan ohjelmistojen turvallisuutta tarjoamalla koulutusta ja avoimen lähdekoodin projekteja ympäri maailman [21].

WhiteSource on *Mend.io*:n vanha nimi, mutta selvyysden vuoksi tässä opinnäytetyössä käytetään edelleen vanhaa nimeä, koska tämän opinnäytetyön kohteena olevassa projektissa on edelleen käytössä WhiteSource-tehtävät. WhiteSource on koodianalyysityökalu, joka tarkistaa avoimen lähdekoodin työkaluja ja liitännäisiä. Se varoittaa tietoturvariskeistä ja tarvittaessa päivittää automaattisesti vanhentuneet versiot. Lisäksi WhiteSource auttaa yrityksiä noudattamaan käyttämiensä avoimen lähdekoodin komponenttien lisenssejä sekä tunnistaa koodin laadun ongelmia, kuten code smellejä ja ohjelmointivirheitä. Se tukee lähes kolmeakymmentä eri ohjelmointikieltä ja on integroitavissa useimpiin DevOps-alustoihin. [22.]

3 Microsoft Azure

Azure on Microsoftin tarjoama julkinen pilvipalvelualusta. Se tarjoaa satoja palveluita IT-infrastruktuurin tuottamiseen ja hallintaan sekä sovellusten kehittämiseen. Sen tarjoamia palveluita ovat muun muassa erilaiset tiedon tallennusratkaisut kuten *Blob Storage* ja *Cosmos DB*, virtuaaliverkot, olemassa olevien sovellusten isännöinti virtuaalikoneilla ja uusien sovellusten kehitys ja julkaisu esimerkiksi *Azure App Servicessä*, tekoäly- ja koneoppimispalvelut sekä monet muut. Azure tarjoaa myös runsaasti tukityökaluja palveluiden tilan ja kustannusten seurantaan. [23.]

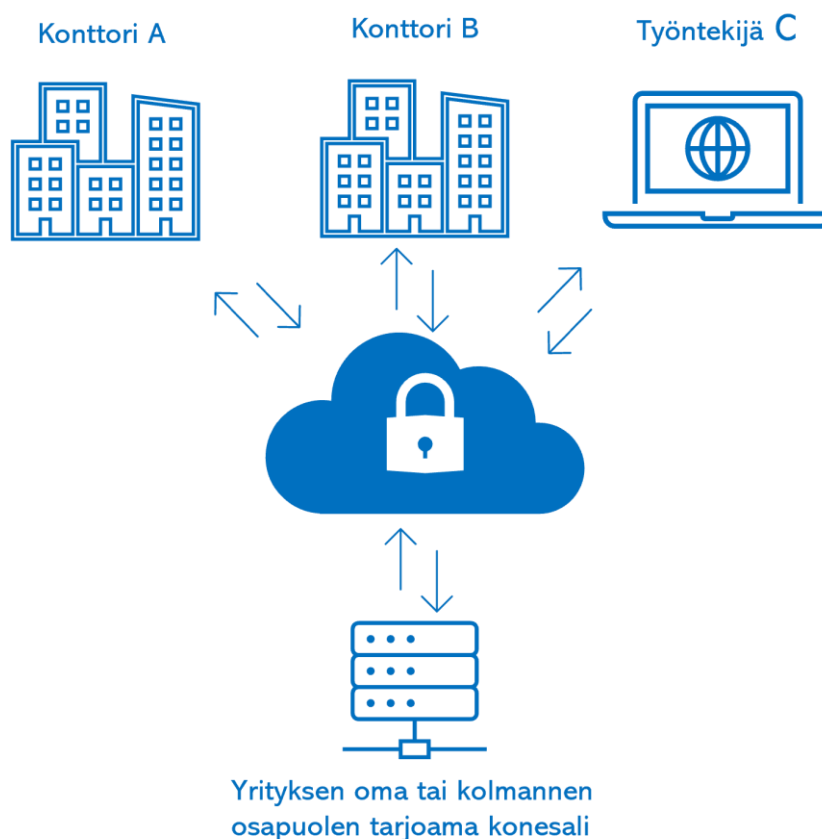
3.1 Pilvipalveluista yleisesti

Pilvipalveluilla tarkoitetaan tietoteknisten palveluiden tarjoamista Internetin välityksellä. Kun palvelut toimitetaan Internetin välityksellä, välttyy palvelun käyttäjä fyysisen infrastruktuurin järjestämiseltä ja ylläpidolta niin halutessaan. Tästä johtuen pilvipalveluiden hyödyntämisen etuna on nopea skaalautuvuus, kun uusien palvelimien tai konesalien pystyttämistä ei tarvitse odottaa, sekä jaettu vastuu pilvipalvelun tarjoajan kanssa. [24.] Toisin kuin oman IT-infrastruktuurin perustamisessa, pilvipalveluiden käyttöön ei tarvita suurta etukäteen maksettavaa summaa, vaan niistä maksetaan käyttöasteen mukaan. Palvelun kysynnän kasvassa resurssit saa parhaimmillaan skaalautumaan automaattisesti ylöspäin tarpeen mukaan. Resursseista pääsee myös nopeasti eroon, kun niille ei ole enää tarvetta, jolloin turhaa maksettavaa esimerkiksi käyttämättömistä palvelimista ei synny. [25.]

Tarjolla on useita pilvipalveluiden tuottajia, joista suurimpia ja tunnetuimpia ovat Amazon Web Services (AWS), Google Cloud sekä Microsoft Azure, joka on käytössä tämän opinnäytetyön kohteena olevassa projektissa.

Pilvipalveluita on mahdollista hyödyntää tarpeen mukaan. Koko IT-infrastruktuuri voi olla pilvessä (julkinen pilvi), tai mikäli esimerkiksi asiakkaan tietosuojavaatimukset edellyttävät tiedon säilyttämistä tietyn maan rajojen sisäpuolella, on

mahdollista yhdistää pilvipalveluita ja omaa infrastruktuuria (hybridi-pilvi) tai kuvassa 5 esitettyä täysin yksityistä pilveä, joka on pystyssä joko yrityksen omassa konesalissa, sen toisessa toimipisteessä tai kolmannen osapuolen tarjoamana yritykselle omistetussa konesalissa.



Kuva 5. Esimerkkikuvaus yrityksen yksityisestä pilvestä.

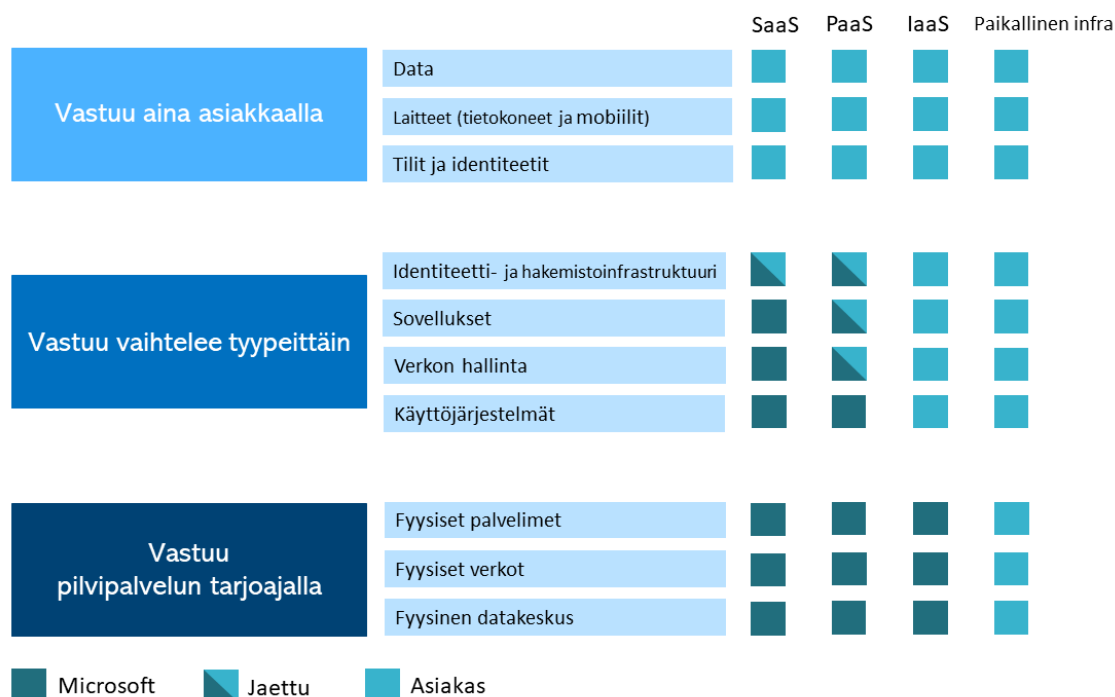
Lisäksi yhä yleistynyt ratkaisu on yhdistellä palveluita eri pilvipalvelun tarjoajilta. (*multi-cloud*). [26.]

Vastuu asiakkaan ja palveluntarjoajan välillä jakaantuu eri tavalla riippuen pilvipalvelun tyypistä. Tyyppejä on kolme: infrastruktuuri palveluna (Infrastructure as a Service eli IaaS), alusta palveluna (Platform as a Service eli PaaS) sekä sovellus palveluna (Software as a Service eli SaaS).

Infrastruktuuri palveluna on asiakkaan kannalta mukautuvien, mutta myös eniten vastuuta sisältävä palvelutyyppi, jossa pilvipalvelun tarjoaja on ainoastaan vastuussa palvelimista, niiden verkkoyhteyksistä ja fyysisestä turvallisuudesta. Asiakas taas on vastuussa kaikesta muusta käyttöjärjestelmän asennuksesta, määrittelystä ja ylläpidosta lähtien. Asiakas käytännössä vuokraa palvelimia pilvikonesalista ja määrittää itse niiden käytön. [27.]

Alusta palveluna tarjoaa asiakkaalle alustan kehittää ja ylläpitää sovelluksia ja palveluita. PaaS:ssa pilvipalvelun tarjoajan vastuisiin kuuluu fyysisen palvelimen, muistin ja verkkoyhteyden ylläpidon lisäksi esimerkiksi sovelluksen kehitykseen, testaamiseen, julkaisuun sekä liiketoiminta-analytiikkaan käytettävien työkalujen ja sovelluskehysten tarjoaminen. [28.]

Sovellus palveluna on kaikista kattavin pilvipalvelutyyppi, jossa asiakas käytännössä vuokraa tai käyttää valmista sovellusta, kuten sähköpostia tai viestisovellusta. SaaS mahdollistaa vähiten palvelun yksilöintiä asiakkaalle, mutta se on myös helpoin ottaa käyttöön. Asiakas on ainoastaan vastuussa datasta, jota sovellus käyttää, sovellusta käyttävien laitteiden yhteyksistä sekä käyttäjien pääsyoikeudesta sovellukseen. Pilvipalvelun tarjoajan vastuulle jää lähes kaikki muu: palvelimien fyysinen turvallisuus, ylläpito ja verkkoyhteydet sekä tarjolla olevien sovellusten kehitys ja ylläpitäminen. [29.]



Kuva 6. Eri tehtävät jakaantuvat pilvipalvelun tarjoajan ja asiakkaan kesken pilvipalvelutyypistä ja paikallisesta IT-infrastruktuurista riippuen. [29.]

3.2 Azure DevOps

Vuonna 2001 17 ohjelmistoalan merkittävää henkilöä, mukaan lukien tässä opinnäytetyössä aiemmin mainitut Martin Fowler, Robert C. Martin ja Kent Beck, allekirjoittivat niin kutsutun Ketterän sovelluskehityksen julistuksen, *The Agile Manifeston*. Julistus sisältää kaksitoista ketterän sovelluskehityksen periaatetta, joiden tarkoitus oli tarjota vaihtoehto vallalla olleelle, kankealle ja dokumentointiorientoituneelle sovelluskehitykselle. Konkreettisten ideoiden lisäksi julistuksella haluttiin tuoda esiin myös syvällisempää kulttuurin ja arvojen muutosta, jossa painotetaan ihmisten, luottamuksen ja yhteistyön tärkeyttä. [30.]

Ketterällä sovelluskehityksellä tarkoitetaan iteratiivista sovelluskehitystä ja projektinhallintaa, eli sovellusta kehitetään pienissä osissa ja asiakas pidetään mukana koko kehitystyön ajan antamassa palautetta ja esittämässä uusia ominaisuuksia. Se usein nopeuttaa sovelluksen käyttöönottoa ja suurin etu on, mikäli sovellus ei vastaakaan asiakkaan toiveita. Tästä saadaan tieto jo varhaisessa

vaiheessa ja sovellusta voidaan muuttaa asiakkaan toiveiden mukaiseksi. Valmiin lopputuloksen ja asiakkaan toiveiden kohtaamattomuus voi olla ongelma esimerkiksi vesiputouskehitysmallissa, jossa suunnittelu tehdään huolellisesti projektin alussa ja sen pohjalta kehitystiimi tuottaa valmiin sovelluksen, jonka asiakas näkee vasta projektin päätyttyä. [31.]

Ketterän sovelluskehityksen yleistyttyä 2000-luvun alun jälkeen kuitenkin havduttiin siihen, että tuotantotiimi, jonka vastuulla on sovelluksen käyttöönotto ja ylläpito, jäi huomiotta uudessa kehitysmenetelmässä. Tästä syntyi DevOps, joka tuo yhteen sekä kehitys- että tuotantotiimin ja virtaviivaistaa näiden kahden yhteistyön. [31.]

DevOpsin tavoitteena on parantaa työskentelyä koko sovelluskehityksen elinkaaren läpi ja näin ollen nopeuttaa organisaation sovellusten ja palveluiden julkaisua. Sen peruspilareita ovat ketterä kehitys, automaatio sekä luottamuksen ja yhteenkuuluvuuden lisääminen kehittäjien ja ylläpitäjien välillä. DevOps tulee sanoista *development* eli *dev*, ohjelmistokehitys ja *operations* eli *ops*, tuotanto. [31.]

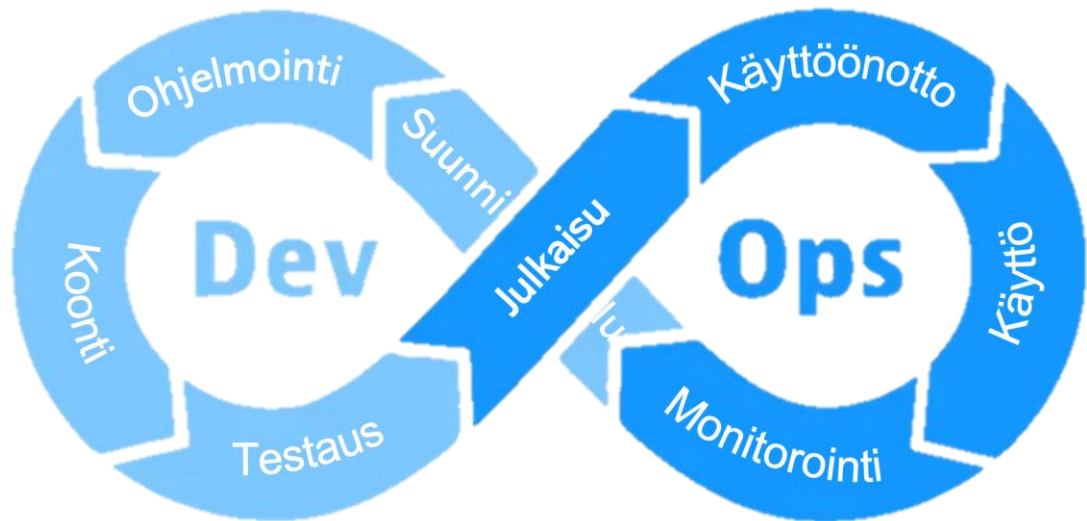
DevOpsin keskeisiä periaatteita ovat *The Three Ways* ja *CALMS*. The Three Ways on Gene Kimin, Kevin Behrin ja George Spaffordin kirjasta *The Phoenix Project* tutuksi tullut kolmen kohdan periaate, joista ensimmäinen on ”ison kuvan tarkastelu” sen sijaan, että keskitytään yksittäisen työntekijän, tiimin tai osaston suoritukseen. Organisaation tulisi sen sijaan keskittyä luomaan virtaviivaisia työnkulkuja yrityksen, kehitystiimin, tuotantotiimin ja asiakkaan välille. Toinen periaate on nopea palautteen saaminen ja sen hyödyntäminen työn laadun parantamisessa. Kun palautetta saadaan usein, tieto säilyy tiimillä henkilöstömuutoksista huolimatta sekä myös virheet huomataan ja korjataan nopeammin. Kolmantena periaatteena on yrityksen kulttuurin muutos kokeiluun ja jatkuvaan oppimiseen kannustavaksi, hyväksyen riskit ja tehden virheistä osan oppimisprosessia. [32.] CALMS tulee sanoista kulttuuri (*Culture*), jolla tarkoitetaan tiimien välistä yhteistyötä, automaatio (*Automation*) lisää nopeutta ja laatua työnkulkuun, *Lean*-periaatteet [33], eli:

- turhan ja tarpeettoman työn eliminointi
- jatkuva oppiminen
- päätösten teko mahdollisimman myöhäisessä vaiheessa
- nopea toimittaminen
- tiimin voimaannuttaminen
- laadun rakentaminen
- kokonaisuuden tarkastelu,

jatkuva mittaus (*Measurement*) prosessien parantamiseksi sekä jakaminen (*Sharing*), eli yhteistyön korostaminen. [32.]

DevOpsille keskeisiä teknologioita ovat jatkuvan integraation ja jatkuvan toimitamisen työkalut, joiden avulla automatisoidaan muun muassa laadun varmistusta, testaamista ja julkaisua sekä reaaliaikainen seuranta, tapahtumien ja konfiguraatiohallinta sekä yhteistyöalustat. Lisäksi nykyään usein myös pilvipalvelut, mikropalvelut eli pieniksi osiksi jaetut palvelut, joista sovellus sekä kontitus, eli koodin ja riippuvuuksien paketointi, jotta sovellus toimii nopeasti ja vakaasti ympäristöstä riippumatta. [34; 35; 36.]

DevOpsia usein kuvataan päättymättömänä silmukkana, joka lähtee liikkeelle suunnittelusta ja etenee ohjelmointiin, koontiin, testaukseen, julkaisuun, käyttöönnottoon, käyttöön, monitorointiin ja palautteen kautta jälleen suunnitteluun, josta silmukka lähtee uudelleen käyntiin kuvan 7 mukaisesti.



Kuva 7. DevOps kuvattuna päättymättömänä silmukkana. [35]

Vaikka kuvaaja on varsin geneerinen, se kertoo hyvin, mistä DevOpsissa on kyse, eli jatkuvasta ja saumattomasta yhteistyöstä tiimien välillä.

Azure DevOps sisältää kaikki sovelluksen kehitykseen ja ylläpitoon vaadittavat toiminnot: työn suunnittelu ja seuranta tapahtuu *Azure Boards* -palvelulla ketteriä menetelmiä hyödyntäen; *Azure Repos* tarjoaa joko Git- tai TFVC-versionhallinnan; *Azure Artifacts* auttaa pakettien ja riippuvuuksien hallinnoinnissa; *Azure Test Plans*in avulla kehittäjät voivat suunnitella, ajaa ja seurata sovellustensa testausta sekä *Azure Pipelines*, jonka avulla voidaan toteuttaa jatkuvan integraation ja -toimittamisen (CI/CD) putkia. Nämä palvelut on suunniteltu toimimaan saumattomasti, ja ne tukevat useita eri ohjelmointikieliä, alustoja ja ohjelmistokehyksiä. Tässä opinnäytetyössä keskitytään pääasiassa *Azure Pipelines* -palveluun, sillä sen sisältämät ominaisuudet ovat keskeisimmät opinnäytetyön osalta. [36.]

Azure DevOpsista on olemassa kaksi versiota: *Azure DevOps Services*, joka on täysi pilvipalvelu, sekä konesalissa sijaitseva *Azure DevOps Server*. Molemmissa on samat perusominaisuudet, mutta *Services* tarjoaa lisäksi yksinkertaisemman palvelinhallinnoinnin, paremman yhteyden eri toimipisteiden välillä

sekä välittömän pääsyn uusimpiin ominaisuuksiin. Server voi kuitenkin olla käyttäjälle välttämätön, mikäli ei ole mahdollisuutta tietosuojasäännösten vuoksi säilöä dataa oman verkon ulkopuolelle. [36.]

3.2.1 Azure Pipelines

Azure Pipelines on keskeinen osa Azure DevOpsia. Sen avulla kehitystiimit automatisoivat sovelluskehityksen prosesseja. Se yhdistää jatkuvan integraation, toimittamisen ja testauksen automaattiseksi koonniksi, testaukseksi ja toimittamiseksi. Azure Pipelinesin käytön vaatimuksena on, että lähdekoodi on versionhallinnassa. Azure DevOps tukee Git- ja Azure Repos -versionhallintatyökaluja. Se tukee useita eri ohjelmointikieliä ja testauskehyksiä ja putkiin on mahdollista yhdistää myös komentokehotteita sekä PowerShell- ja Shell-skriptejä. [37.]

Jatkuvan integraation tehtäviä voidaan automatisoida suorittamalla koonti ja testaus koodin yhdistämisen yhteydessä ja asettaa esimerkiksi näiden onnistunut suoritus ehdoksi uuden koodin päähaaraan yhdistämiselle. Onnistuneen koonnin ja testauksen jälkeen voidaan putkeen lisätä automaattinen artefaktin luonti. Artefaktilla tarkoitetaan kokoelmaa tiedostoja tai paketteja, jotka vaaditaan sovelluksen ajamiseen kuten .exe-, tai .dll-päätteiset tiedostot tai NuGet-, npm- ja Maven-paketit. Näitä artefakteja hyödynnetään myöhemmin jatkuvan toimittamisen putkissa. [37.]

Jatkuva toimittaminen on prosessi, joka kokoaa, testaa ja ottaa sovelluksen käyttöön eri testi- ja tuotantoympäristöissä, kuten Windows- ja Linux-ympäristöissä. Kun koodi testataan useissa eri ympäristöissä, se lisää laatua, kun voidaan olla varmoja, että sovellus toimii toivotulla tavalla esimerkiksi eri käyttöjärjestelmillä. Tämän jälkeen syntyneet käyttöönottokelpoiset artefaktit voidaan julkaista osana automaattista prosessia uusina versioina ja päivityksinä olemassa oleviin sovelluksiin. [37.]

Azure Pipelines on siirtymässä vanhasta graafisesta käyttöliittymästä kohti YAML-putkia. YAML on suosittu, helposti luettava merkintäkieli, jota käytetään

usein konfiguraatiotiedostoissa. Sen nimi on lyhenne joko sanoista ”*Yet another markup language*”, vapaasti käännettynä ”*Taas yksi merkintäkieli*” tai ”*YAML ain’t markup language*”, ”*YAML ei ole merkintäkieli*”. YAML:ssa on ominaisuuksia useista eri ohjelmointikielistä, ja se on JSON:in ylijärjestelmä, eli se sisältää omien ominaisuuksiensa lisäksi kaikki JSON:in ominaisuudet, jolloin JSON on validi kieli myös YAML-tiedostoissa. [38.] YAML-tiedosto kulkee projektin lähdekoodin mukana, ja se löytyy usein projektin juurihakemistosta tai mahdollisesti yhdelle tai useammalle putkelle luodusta kansioista.

YAML:n sisäkkäisiä rakenteita kuvataan Pythonin tapaan sisennyksillä. Toisin kuin Python, YAML ei hyväksy sarkaimen käyttöä, vaan sisennykset on tehtävä välilyönnein. YAML:iin voi lisätä kommentteja aloittamalla ne #-merkillä. [38.]

Kaikilla putkilla on samat perustoiminnallisuudet: *trigger* käynnistää putken ja sille voi määrittää erilaisia arvoja, kuten esimerkiksi tietyn haaran päivityksen. Haluttu tarkka ajankohta tai jonkin aiemman koonnin onnistunut ajo voidaan asettaa putken käynnistäjäksi. Triggeriksi voi määrittää myös koodiesimerkki 1:n mukaisesti *none*, joka on tarpeen esimerkiksi luotaessa PR-validointiputkea. Tällöin käynnistys tapahtuu päähaaralle asetettavan haarasäännön (*branch policy*) asetuksista. [39.]

```
# Päähaaran päivitys käynnistää putken
trigger:
- master
```

Koodiesimerkki 1. Päähaaraan (*master*) tehtävät muutokset käynnistävät putken.

Stage eli vaihe määrittää putken eri vaiheille rajat. Jokaisessa putkessa on vähintään yksi vaihe, mutta putken voi tarvittaessa jakaa useampaan, esimerkiksi koontiin, testaukseen ja käyttöönottoon. Oletusarvoisesti vaiheet ajetaan määrittelyjärjestyksessä yksi toisensa jälkeen, mutta niitä voi myös ajaa yhtäaikaaisesti, tai asettaa niille riippuvuuksia ja ehtoja liittyen muihin vaiheisiin. Koodiesimerkissä 2 on asetettu erilaisia ehtoja ja riippuvuuksia, jotka määräävät vaiheiden ajojärjestyksen. [40.]

```

stages:
- stage: A

# vaihe B ajetaan vain, jos vaihe A epäonnistuu
- stage: B
  condition: failed()

# vaihe C ajetaan, jos A onnistuu
- stage: C
  condition: succeeded('A')

# vaihe D-1 ajetaan C:n jälkeen
- stage: D-1
  dependsOn: C

# vaihe D-2 ajetaan rinnakkaisesti D-1:n kanssa
- stage: D-2
  dependsOn: C

# vaihe E ajetaan D-1 ja D-2 vaiheiden jälkeen
- stage: E
  dependsOn:
    - D-1
    - D-2

```

Koodiesimerkki 2. Vaiheiden ajojärjestystä voidaan muuttaa ehdoilla ja riippuvuuksilla.

Stage sisältää vähintään yhden tai useamman *jobin* eli työn. Työ on sarja askeleita (*step*), jotka ajetaan yksi kerrallaan osana kokonaisuutta. Mikäli töitä on vain yksi, ei sitä tarvitse erikseen määritellä, mutta jos työllä haluaa määritellä tarkemmin ominaisuuksia tai lisätä useamman työn, pitää se määritellä koodiesimerkki 3:n mukaisesti YAML-tiedostoon. Töille on myös mahdollista määritellä ehtoja ja riippuvuuksia kuten vaiheillekin, mutta putken tulee sisältää vähintään yksi työ ilman näitä. [40.]

```

jobs:
# työ A ajetaan Microsoftin isännöimällä agentilla, jossa on Ubuntun
# viimeisin versio ja tulostaa "Suoritetaan työ A"
- job: A
  pool:
    vmImage: 'ubuntu-latest'
  steps:
    - bash: echo "Suoritetaan työ A"

# työ B ajetaan Microsoftin isännöimällä agentilla, jossa on Windowsin
# viimeisin versio ja tulostaa "Suoritetaan työ B"
- job: B
  pool:
    vmImage: 'windows-latest'
  steps:
    - bash: echo "Suoritetaan työ B"

```

Koodiesimerkki 3. Esimerkkejä erilaisista töistä ja niiden määrittelyistä.

Töitä voi olla erilaisia riippuen siitä, missä niitä ajetaan, esimerkiksi agenttien, palvelimen tai konttien työt. Työ voi myös olla agentiton, jolloin se voi esimerkiksi käynnistää Azure Function tai REST API -tehtävän. Yleisimpiä ja tässäkin opinnäytetyössä käytettyjä ovat agenttireservien työt. Agenteilla tarkoitetaan joko asiakkaan itse tai Microsoftin isännöimiä virtuaalikoneita, jotka suorittavat putken määräämät työt. Itse isännöidyille agenteille voi asettaa vaatimuksia tietystä ominaisuuksista, jotka ovat edellytyksenä ajon onnistumiselle. Vaatimuksena voi esimerkiksi olla tietty käyttöjärjestelmä tai että jokin tietty ohjelma, kuten Visual Studio on asennettuna agentille. Mikäli agenttireservissä ei ole vapaana putken vaatiman määrittelyn mukaista agenttia, jää putki odottamaan sopivan agentin vapautumista. Kun työ käynnistetään, syntyy agentille työtila, eli hakemisto, johon se lataa lähdekoodin, ajaa pyydetyt tehtävät ja / tai skriptit ja tallentaa niiden lopputuloksen. Tämän hakemiston alle syntyy muita hakemistoja esimerkiksi lähdekoodille, artefakteille ja testituloksille. [41.]

Askel eli *step* on putken pienin osa. Askeleita voivat olla esimerkiksi koonti ja testaus, ja se voi olla joko skripti tai tehtävä eli *task*. Azure Pipelines tarjoaa useita valmiita tehtäviä esimerkiksi testaukselle ja koonnille helpottamaan putken luontia, mutta on myös mahdollista luoda omia tehtäviä. Tehtävät ovat versioituja ja versio on aina määritettävä tehtävän yhteydessä lisäämällä @x tehtävän nimen loppuun. Mikäli tehtävään tulee uusi versiopäivitys, kuten päivitys

1.0-versiosta 2.0-versioon, tulee sen numero päivittää manuaalisesti tehtävän yhteyteen. Pienempien päivitysten yhteydessä, kuten 1.7-versiosta 1.8-versioon, putki osaa käyttää uusinta versiota, eikä tätä tarvitse päivittää putken tehtävään manuaalisesti. [41.]

```
steps:
# Azure Pipelinesista valmiina löytyvä tehtävä
- task: PublishTestResult@2

# esimerkki skriptistä, joka on yksilöity vain kyseistä putkea koskeva
askel
- script: echo "Hello world"

# PowerShell-tehtävä, jonka voi lisätä omana tiedostona repositorioon
# ja kutsua sitä putkesta
- task: PowerShell@2
  inputs:
    targetType: 'filePath'
    filePath: 'skriptitiedosto.ps1'

# Powershell-tehtävä, joka sisältää lyhyen skriptin, joten sen voi
# lisätä suoraan putkeen
- task: PowerShell@2
  inputs:
    targetType: 'inline'
    script: Write-Host "Hello world"
```

Koodiesimerkki 4. Esimerkkejä Azure Pipelinesta löytyvästä sekä skripti- ja PowerShell-tehtävistä.

Koodiesimerkki 5 havainnollistaa putken eri vaiheiden hierarkian. Kun töitä tai tehtäviä on vain yksi, ei tarvita erikseen ylätasoa määrittelyä *jobs:* tai *steps:*. Käytettävät agentit (*pool*) on mahdollista määrittää koko putkelle koodiesimerkki 5:n mukaan, vaiheelle tai työlle.


```

trigger: none
pool:

stages:
- stage: A
  jobs:
  - job: A.1
    steps:
    - task: A.1.1
    - task: A.1.2
  - job: A.2
    - task: A.2.1
- stage: B
  - job: B.1
    - task: B.1.1
    - script: B.1.2

```

Koodiesimerkki 5. YAML-putken hierarkia.

Azure Pipelines tukee kaikkia ohjelmointikieliä ja -alustoja. Sen avulla on mahdollista automatisoida projektien koonti helposti ja turvallisesti. Lisäksi se on ilmainen avoimen lähdekoodin projekteille. [37.]

4 Pull request -validointiputki

Opinnäytetyön aihe tuli kyseisen projektin kehitystiimiltä, ja sen määrittely toteutettiin tiimin esimiehen ja tiimin arkkitehtien kanssa. Projektissa oli aiemmin käytössä ainoastaan päähaaraa vasten ajettavia putkia, joissa ajettiin muun muassa testit ja laatuanalyysityökalut. PR:n hyväksymiselle oli asetettu ehdoiksi kahden tiimiläisen koodikatselmointi sekä kommenttien hyväksytty selvitys. Ihmisen tekemä koodikatselmointi voi viedä aikaa eikä ihminen välttämättä huomaa kaikkia ongelmakohtia esimerkiksi kiireen tai tietämättömyyden vuoksi. Näitä ehtoja ei koettu riittäviksi, ja erityisesti analyysityökalujen, kuten SonarQuben tulosten ja testien läpimenon seuranta haluttiin helpottaa. Tulokset haluttiin saada suoraan Azureen sen sijaan, että niitä pitäisi käydä lukemassa analyysityökalujen omilta sivuilta.

Projektille toivottiin jatkuvan integraation putkea osaksi PR:ää, jonka onnistunut läpäisy asetettaisiin vaatimukseksi, ennen kuin PR voidaan hyväksyä ja uusi koodi yhdistää osaksi päähaaraa. Putken tulisi sisältää koonti, yksikkötestaus,

SonarQube- ja WhiteSource-analyysit sekä 80 prosentin testikattavuus uudelle koodille. Testikattavuudella tarkoitetaan sitä prosentuaalista tasoa, jolla lähdekoodin kaikki rivit on testattu automaattisen testauksen yhteydessä. Sen avulla voidaan havaita, mikäli jokin osa koodista ei ole kattavasti testattu, ja näin vaatia kehittäjää parantamaan testikattavuutta. Näiden ratkaisujen tavoitteena oli vähentää inhimillisiä virheitä ja puuttua ongelmakohtiin ennen niiden päähaaraan pääsyä.

4.1 Projektin tiedot

Projekti hyödyntää .NET Framework 4.7.2 -ohjelmistokehystä. .NET (aiemmin .NET Core) on Microsoftin kehittämä kehitystyökalujen kokoelma eri alustoille, kuten Windowsille, Linuxille, macOS:lle sekä iOS:lle ja Androidille. Lisäksi kehysperheeseen kuuluu Xamarin/Mono, joka on erikseen suunnattu kaikille isoille mobiilikäyttöjärjestelmille. .NET Framework on alkuperäinen, Windowsille suunnattu toteutus.

Projektissa käytetty .NET Framework -versio on vanha, eivätkä monet laajennukset enää tue tätä versiota. Projekti onkin läpikäymässä migraatiota uudempaan versioon, .NET 6:een. Uusi koodi on tarkoitus toteuttaa jatkossa .NET 6 -versiolla, ja vanha koodi siirretään käyttämään geneeristä monikohdekehystä (multi-target framework). Tällöin voidaan edelleen hyödyntää vanhan koodin käyttämiä .NET Framework 4.7.2 -ohjelmointirajapintoja sekä uuden koodin .NET 6 -ohjelmointirajapintoja. [42.]

4.2 Tiimin haastattelu

Lähtötilanteen selvittämiseksi kysyin seitsenhenkiseltä kehitystiimiltä seuraavat kysymykset, joihin pyysin mahdollisimman rehellisiä vastauksia. Vastaukset jätettiin anonymistiksi. Tiimiin oli kysymysten esityksen aikaan juuri liittynyt muutamia uusia jäseniä, jotka eivät kokeneet voivansa vastata lyhyen työskentelyaikansa vuoksi moniin kysymyksiin merkityksellisesti.

1. Jos mietit kymmentä viimeisintä PR:ää, kuinka usein ajoit yksikkötestit?

Vastaukset vaihtelivat 0 ja kymmenen välillä, alle viiden vastausten kuitenkin olleen yleisin. Vaikka yksikkötestit eivät automaattisesti tarkoita, että koodi on ongelmatonta, on laadukkailla ja kattavilla yksikkötesteillä mahdollista pienentää riskiä toimimattoman koodin päätyemisestä päähaaraan. Automatisoimalla tämän osaksi PR:ää on mahdollista pienentää riskiä entisestään, kun ajon muistaminen ei ole ihmisen varassa.

2. Jos mietit viimeisintä vuotta, kuinka usein päähaara / käyttöönnotot / putket eivät ole onnistuneet? Voit mainita myös arvion näistä aiheutuneista viivästyksistä.

Vastaukset kysymykseen vaihtelivat reilusti. Arvioita tuli noin viidestä kymmeneen, jotka kestivät yhden arvioin mukaan noin kolme tuntia per ongelmatilanne. Eräs vastaaja muisteli viime kesänä tapahtuneen ongelmatilanteen kestäneen viikon, kun paikalla olleet tiimin jäsenet eivät osanneet ratkaista ongelmaa. Toinen vastaaja arvioi päähaarassa olleen ongelmia jatkuvan toimittamisen putkien kanssa viidestä kymmeneen kertaa.

3. Jos mietit viimeistä vuotta, kuinka usein olet tarkistanut SonarQube-analyysin tulokset liittyen koodimuutoksiisi?

Suurin osa kehittäjistä myönsi, etteivät ole koskaan käyneet katsomassa analyysin tuloksia SonarQube-palvelusta. Monet huonot koodikäytännöt ovat siis jääneet huomaamatta ennen päähaaraan yhdistämistä, mikä on kerryttänyt kohutuullisen määrän teknistä velkaa.

4. Jos mietit viimeistä vuotta, kuinka usein olet tarkistanut WhiteSource-analyysin tulokset liittyen koodimuutoksiisi?

Suurin osa kehittäjistä vastasi tähänkin, etteivät ole kertaakaan tarkistaneet tuloksia yhtä vastaajaa lukuun ottamatta. Hänen arvionsa oli noin kymmenen kertaa.

5. Oletko koskaan tehnyt korjauksia koodiisi SonarQube-analyysin löytämien havaintojen perusteella sen jälkeen, kun PR on jo hyväksytty?

Kahta vastaajaa lukuun ottamatta vastaukset olivat kielteisiä, kuten voi olettaa jo analyysin tarkastelun pienestä määrästä.

6. Oletko koskaan tehnyt korjauksia koodiisi WhiteSource-analyysin löytämien havaintojen perusteella sen jälkeen, kun PR on jo hyväksytty?

Tähän ainoastaan yksi kehittäjä vastasi myöntävästi. Kysymykset liittyen WhiteSourceen eivät olleet erityisen hyödyllisiä, sillä perehdyttyäni enemmän sen toimintaan, WhiteSource perustuu hälytyksiin, joita se lähettää havaitessaan ongelmallisia liitännäisiä eikä näin ollen varsinaisesti vaadi erillistä tulosten seuranta kehittäjältä itseltään.

Kysely antoi käsityksen, että projektia on tehty toiminnallisuus edellä, eikä aikaa tai halua ole aiemmin ollut kehittää jatkuvan integraation ratkaisuja. Erityisesti yksikkötestauksen vähyys voi olla iso syy, miksi projektissa on ollut jonkin verran koko kehitystiimin pysäyttäviä ongelmia päähaarassa.

4.3 WhiteSource (Mend)

Tarkoituksena oli lisätä myös WhiteSource-analyysi PR-validointiputkeen. Asiaan perehdyttäessä kävi kuitenkin ilmi, että WhiteSource-analyysi ei tue haaroille tehtävää analyysiä. Tämä tarkoittaa, että aina kun analyysi ajetaan uudelleen eri haarasta, edellinen tulos ylikirjoitetaan eikä näin ollen aiemman haaran analyysistä jää jälkeä. Tätä ei pidetty tiimin kannalta toimivana ratkaisuna, sillä kehitystyö on nopeatempoista ja uusia pull requesteja tulee useita päivän aikana. Tästä syystä päädyimme jättämään WhiteSource-analyysin pois PR-validointiputkesta, ja se ajettaisiin jatkossakin osana päähaaraa vasten ajettavaa putkea, joka on aiemmin huolehtinut koonnista, testauksesta, SonarQube- ja WhiteSource-analyyseistä.

WhiteSource olisi mahdollista ottaa osaksi PR-validointia, mikäli koko Azure DevOps siirrettäisiin palvelimelta Microsoftin tarjoamaan Azure DevOps Service -palveluun. Tällöin WhiteSource on mahdollista integroida osaksi Azure Repos -palvelua: analyysi ajettaisiin, mikäli se havaitsisi muutoksia NuGetin, eli .NET-sovelluksen pakettienhallintaohjelman määrittelytiedostossa tai projektiin lisättäisiin uusi tai poistettaisiin lähdekooditiedosto, joka sisältää WhiteSourcen tukeman laajennuksen. Mikäli analyysi ajetaan ja se havaitsee riskejä, WhiteSourcen botti luo automaattisesti Azure DevOpsin Boards-ominaisuuteen uuden tehtävän *Work Items* -kohdan alle ja määrittää riskin tason. WhiteSourcen analyysi on myös mahdollista lisätä *Status Check*:si, jolloin sen onnistunut ajo on ehtona PR:n läpimenoille. Lisäksi analyysin tuloksia on mahdollista tarkastella WhiteSourcen portaalissa, koodinlisäysten eli *commit*ien yhteyteen syntyvistä kommenteista sekä sähköposti-ilmoituksista. Nämä ominaisuudet ovat kuitenkin tarjolla ainoastaan Azure DevOps Servicessä. [43.]

4.4 SonarQube

SonarQube oli jo ennestään käytössä projektin muissa putkissa, joten sen liittäminen PR-validointiputkeen vaikutti lähtökohtaisesti mutkattomalta. Liittämisestä ei tarvinnut asentaa ADO:iin eikä tietovarastoa lisätä SonarQubeen enää uudelleen. Näin ollen Azure Pipelinesistä löytyi tarvittavat tehtävät, jotka on jaettu kolmeen vaiheeseen: *Prepare*, *Analyze*, *Publish*, eli valmistautuminen, jossa annetaan käytettävän tietovaraston nimi ja tiliavain; analyysi, joka suorittaa nimensä mukaisesti analyysin koodille sekä analyysin tulosten julkaisu SonarQubeen.

Kuitenkin lähemmin tarkasteltuani SonarQubea havaitsin, että analyysin tulos kattoi koko projektin koodin eikä kyseisen uuden koodin tuomia muutoksia. Tämä ei ollut tarkoituksen mukaista, vaan tavoitteena oli saada aikaan analyysi, joka koskee ainoastaan uutta koodia, jolloin Samlinkin DevOps-tiimin kanssa asiaa selvitettyä kävi ilmi, että projektin ja SonarQube-palvelimen välillä ei ollut tarvittavaa yhteyttä, jonka vuoksi tiettyjä asetuksia ei ollut mahdollista määrittää SonarQubessa, eikä SonarQube pystynyt esimerkiksi palauttamaan tietoa

takaisin Azureen. Tästä johtuen tuloksia tuli harvoin tarkasteltua, sillä se vaati siirtymistä SonarQube-palveluun.

Palomuuariavausta odotettiin useampi viikko, sillä tämän voi suorittaa tehtävälle omistettu tiimi. Tästä johtuen SonarQuben määrittely ei täysin valmistunut ennen tämän opinnäytetyön valmistumista. SonarQubesta saatiin kuitenkin kommentteja code smelleihin liittyen pull requestin yhteyteen. Koska SonarQube-analyysi ajetaan joka kerta koko projektille, tuli PR:iin mukaan tuhansia kommentteja liittyen myös muuttumattomiin riveihin. Koska projektin branch policyna on kommenttien hyväksytty käsittely, oli SonarQuben huomioivia tiedostoja rajoittava niin, että se ottaa PR-validoinnissa huomioon ainoastaan muuttuneet tiedostot. Tämä toteutettiin PowerShell-skriptin avulla.

4.5 Testikattavuus

Testikattavuus osoittautui kaikista toiminnallisuuksista haastavimmaksi. Tiimin käyttämä ohjelmointiympäristö Visual Studio ei tarjoa testikattavuusraportin luontia kaikissa versioissa, ja näistä Enterprise on ainoa, johon sisältyy testikattavuusraportointi. Monella kehittäjällä, itseni mukaan lukien, on käytössä Visual Studio Professional, joten jouduimme turvautumaan ulkoisiin testikattavuustyökaluihin. Työkalun valinnassa oli otettava huomioon myös SonarQube-yhteensopivuus.

Ensimmäinen tutkittava työkalu oli Coverlet yhdistettynä ReportGenerator-nimiseen työkaluun. Coverletin tehtävä on luoda testikattavuusraportti, ja ReportGenerator muuntaa syntyneen raportin ihmisen luettavaan HTML-tiedostoon, joka on mahdollista julkaista Azure DevOpsissa. Coverlet ei kuitenkaan tukenut vanhaa .NET Framework 4.7.2 -koodia, joten kokeilimme toista vastaavaa työkalua nimeltä OpenCover. Sekä OpenCover että ReportGenerator asennettiin agenteille, ja testikattavuusraportit saatiin luotua testiprojektista ja julkaistua suoraan kuvan 8 mukaisesti Azure DevOpsiin.

#test_1.17 Update pr-pipeline.yml for Azure Pipelines
on Hiekkalaatikko

Run new

...

Summary

Tests

Code Coverage

Extensions

Summary

Generated on:

13.1.2023 - 14.27.28

Parser:

CoberturaParser

Assemblies:

1

Classes:

2

Files:

2

Covered lines:

2

Uncovered lines:

3

Coverable lines:

5

Total lines:

36

Line coverage:

40% (2 of 5)

Covered branches:

0

Total branches:

0

Risk Hotspots

No risk hotspots found.

Coverage

Name	Covered	Uncovered	Coverable	Total	Line coverage	Branch coverage
Test472	2	3	5	36	40% <div></div>	<div></div>
Test472.Class1	2	0	2	21	100% <div></div>	<div></div>
Test472.Program	0	3	3	15	0% <div></div>	<div></div>

Generated by: ReportGenerator 4.5.0

13.1.2023 - 14.27.28

GitHub | www.palmmedia.de

Kuva 8. Testiprojektin suppeuden vuoksi raportti on melko vaatimaton, mutta siitä saa selville lausekattavuuden.

Raportin generointi ei kuitenkaan onnistunut varsinaisessa projektissa ohjelmointikehyksiin liittyvien ongelmien vuoksi. Vaikutti siis siltä, että joko NetStandard tai NetCore-tyyppinen projekti vaaditaan, jotta testikattavuus saadaan toimimaan. Näin ollen vanhan koodin osalta voi olla hyvin vaikeaa saada aikaan testikattavuusraporttia. Testikattavuusraportti vaaditaan, jotta SonarQuben avulla voidaan asettaa quality gate, eli prosenttiraja testikattavuudelle.

4.6 Valmis putki ja sen käyttö

Lopputuloksena syntyi tässä vaiheessa hieman keskeneräinen putki. Seuraavaksi käyn läpi putken vaiheet ja selitän niiden sisällön. Kyseinen putken sisältö on testiprojektista, joten siinä ei ole varsinaisen projektin putkesta löytyvää käsiteltyjen tiedostojen rajausskriptiä.

Koodiesimerkissä 6 triggeriksi on määritetty *none*, sillä putken käynnistys on määritetty branch policyssa. Putkelle on annettu nimeksi *test_1.x*, ja x:n tilalle haetaan automaattisesti versionumero. Pool-kohdassa on määritetty, mitä

agenttien ryhmää putki käyttää. Lisäksi käytettävillä agenteilla on oltava asennettuina DotNetFramework-ohjelmistokehys sekä MSBuild-koontityökalu, joka automatisoi koonnin.

```
trigger: none

name: test_1$(rev:.r)

pool:
  name: 'Cloud2.0'
  demands:
    - DotNetFramework
    - msbuild
```

Koodiesimerkki 6. Putken alustavien tietojen määrittely.

Koodiesimerkki 7:ssä esitellään putken sisältämät muuttujat. Solution-tiedoston pääte on .sln, joten tässä muuttujaksi on määritelty mikä tahansa tiedosto, jolla on kyseinen pääte, eli se on niin kutsuttu villi kortti. Solution-tiedosto sisältää kaikki siihen liittyvät C#-projektit ja niiden kehittämiseen tarvittavat tiedostot ja resurssit sekä tietoa siitä, miten kyseiset tiedostot liittyvät toisiinsa. Koontialustaksi on määritetty mikä tahansa CPU, jolloin CPU määrittyy agentin ominaisuuksien mukaan, mutta sille voi asettaa arvoksi myös jonkin tietyn CPU:n. BuildConfiguration määrittelee, kuinka projektit kootaan ja käyttöön otetaan. Sille on asetettu arvoksi Release, eli koonnin seurauksena syntyy julkaisukelpoinen ohjelma. Lisäksi on määritetty muuttujat agenttien lähdehakemistoon syntyville tiedostoille, joita käytetään myöhemmin PowerShell-skriptissä, jonka avulla luodaan testituloksista testikattavuusraportti.

```
variables:
  solution: '**/*.sln'
  buildPlatform: 'Any CPU'
  buildConfiguration: 'Release'
  TestResultsDirectory: '$(build.sourcesDirectory)\..\TestResults\'
  OutputDirectory: '$(build.sourcesDirectory)\UnitTestRun\'
  coverageDirName: 'TestResultsFromOpenCover'
  reportDirName: 'TestReportsFromReportGenerator'
```

Koodiesimerkki 7. Putken muuttujien määrittely.

Tuntemattomaksi jääneestä syystä koontitehtävä ei löytänyt polkua MSBuild.exe-tiedostoon, joten sille oli kirjoitettava koodiesimerkki 8:n mukainen tehtävä, joka sisältää PowerShell-skriptin. Skripti palauttaa kyseisen tiedoston tarkan sijainnin. Skriptiä kutsutaan PowerShell-tehtävästä (task), ja filePath-syöte kertoo putkelle, mistä kyseinen PowerShell-skriptitiedosto löytyy.

```
steps:
- task: PowerShell@2
  displayName: 'Set MSBuildPath from env var'
  inputs:
    filePath: '$(build.sourcesDirectory)\pipelines\setMSBuildPath.ps1'
```

Koodiesimerkki 8. PowerShell-tehtävä, joka hakee MSBuild.exe.-tiedoston polun.

Koodiesimerkissä 9 olevat tehtävät palauttavat projektien riippuvuudet sekä projektien tarvitsemat työkalut, jotka on määritetty .csproj-tiedostossa. Projektin uudistustöiden myötä ensimmäinen restore-tehtävä toimii vanhan .NET Framework 4.7.2 -kehityksen kanssa, mutta ei uuden .NET 6:en kanssa, joten molemmille kehityksille on tilapäisesti omat restore-tehtävät.

```
- task: NuGetCommand@2
  displayName: 'dotnet restore'
  inputs:
    command: 'restore'
    restoreSolution: '$(solution)'

- task: DotNetCoreCLI@2
  displayName: 'dotnet restore 2.0'
  inputs:
    command: restore
    projects: '$(solution)'
```

Koodiesimerkki 9. Riippuvuuksien ja työkalujen palautus.

Koodiesimerkki 10:ssä asennetaan Visual Studio testausalustan viimeisin vakaa versio agentille.

```
- task: VisualStudioTestPlatformInstaller@1
  displayName: 'VsTest Platform Installer'
  inputs:
    versionSelector: latestStable
```

Koodiesimerkki 10. Testausalustan asennus.

Koodiesimerkissä 11 valmistellaan SonarQube-analyysin luonti. Sille on annettava SonarQube-palvelimen päätepiste, haluttu analyysin ajotapa, joka tässä tapauksessa on MSBuild, SonarQube-projektin yksilöllinen avain, projektin nimi sekä versionumero. Palvelimen päätepiste, projektiavain ja projektin nimi on tässä esimerkissä muokattu projektin yksilöivien tietojen vuoksi. Lisäksi tehtävälle on määritetty lisäominaisuuksia, kuten testiprojektien jättäminen analyysin ulkopuolelle, testiraportin tarkka sijainti, luodun testikattavuusraportin sijainti sekä UTF-8-koodaus. SonarQubePrepare-tehtävä on suoritettava ennen koonti- ja testaustehtäviä.

```
- task: SonarQubePrepare@4
  displayName: 'Prepare analysis on SonarQube'
  inputs:
    SonarQube: 'SonarQube-palvelimen päätepiste'
    scannerMode: 'MSBuild'
    projectKey: 'yksilöllinen projektiavain'
    projectName: 'Projektin nimi'
    projectVersion: '$(Build.BuildNumber)'
    extraProperties: |
      sonar.dotnet.excludeTestProjects=true
      sonar.cs.vstest.reportsPaths=$(TestResultsDirectory)\*.trx
      sonar.cs.opencover.reportsPaths=$(TestResultsDirectory)\$(coverageDirName)\*.html
      sonar.sourceEncoding=UTF-8
```

Koodiesimerkki 11. SonarQube-analyysin valmisteleva tehtävä.

Koodiesimerkissä 12 toteutetaan koonti. Se saa parametreina aiemmin määritettyjä muuttujia, kuten muun muassa solution-tiedoston ja MSBuild.exe-tiedoston sijainnin. Lisäargumentteina on annettu lisäksi hakemisto kootulle projektille sekä kerätään diagnostiikkatietoja koonnista.

```
- task: MSBuild@1
  displayName: 'Build solution'
  inputs:
    solution: '$(solution)'
    msbuildLocationMethod: location
    msbuildLocation: '$(MSBuildPath)'
    platform: '$(BuildPlatform)'
    configuration: '$(BuildConfiguration)'
    msbuildArguments: '/p:OutDir="$(OutputDirectory)\\" /flp:verbosity="Diagnostic"'
    maximumCpuCount: true
    logProjectEvents: true
```

Koodiesimerkki 12. Koontitehtävä.

Koodiesimerkki 13 on tehtävä yksikkötestien ajolle. testAssemblyVer2 määrittää tiedostot, joista testit ajetaan. CodeCoverageEnabled: true sallii testikattavuusdatan keräämisen.

```
- task: VSTest@2
  displayName: 'Test Assemblies'
  inputs:
    testAssemblyVer2: |
      $(OutputDirectory)\*test*.dll
      !**\obj\**
    vsTestVersion: toolsInstaller
    platform: '$(BuildPlatform)'
    configuration: '$(BuildConfiguration)'
    codeCoverageEnabled: true
```

Koodiesimerkki 13. Yksikkötestit ajava tehtävä.

Koska projektissa ei ole käytössä Microsoftin isännöimiä agenteja, joilla on automaattisesti Visual Studio Enterprise asennettuna ja sen myötä testikattavuusdatan keräys, sekä vanhan koodin ollen edelleen .NET Framework 4.7.2, jota esimerkiksi Coverlet-testikattavuustyökalu ei tue, oli tähän tarkoitukseen käytettävä laajennuksia, jotka asennettiin agenteille. Koodiesimerkki 14:n tehtävä PowerShell-skriptille tarkistaa, mikäli agentin työhakemistossa on jo olemassa aiemmasta ajosta testikattavuusraportti, poistetaan se ja luodaan uudet raportit niistä testikokoonpanoista, jotka vastaavat argumenttina annettua hakutermiä. Skriptille annetaan argumenteiksi polku, josta koonnin myötä syntyneet tiedostot löytyvät, hakutermin yhteydessä syntyneille

testikokoonpanoille, raporttityypit Azureen tuotavaa graafista raporttia varten, hakemistot testikattavuusraportille sekä testituloksille.

```
- task: PowerShell@2
  inputs:
    filePath: '$(build.sourcesDirectory)\pipelines\codeCoverage.ps1'
    arguments: -repoDir $(OutputDirectory) -searchPattern "Unit-
Tests472.dll" -reportTypes "HtmlInline_AzurePipelines;Cobertura" -cov-
erageDirName $(coverageDirName) -reportDirName $(reportDirName) -out-
putDir $(TestResultsDirectory)
```

Koodiesimerkki 14. Tehtävä PowerShell-skriptille, joka luo testikattavuusraportin testauksen jälkeen.

Koodiesimerkin 15 tehtävän tarkoitus on julkaista syntynyt testikattavuusraportti Azureen. Syötteeksi annetaan testikattavuustyökalu sekä koostetiedoston sijainti.

```
- task: PublishCodeCoverageResults@1
  displayName: 'Publish code coverage report to Azure'
  inputs:
    codeCoverageTool: 'Cobertura'
    summaryFileLocation: '$(TestResultsDirectory)\$(reportDir-
Name)\Cobertura.xml'
```

Koodiesimerkki 15. Testikattavuusraportin julkaisu Azureen.

Koodiesimerkki 16 käynnistää aiemmin putkessa määritellyn SonarQube-analyysin.

```
- task: SonarQubeAnalyze@4
  displayName: Run SonarQube Analysis
```

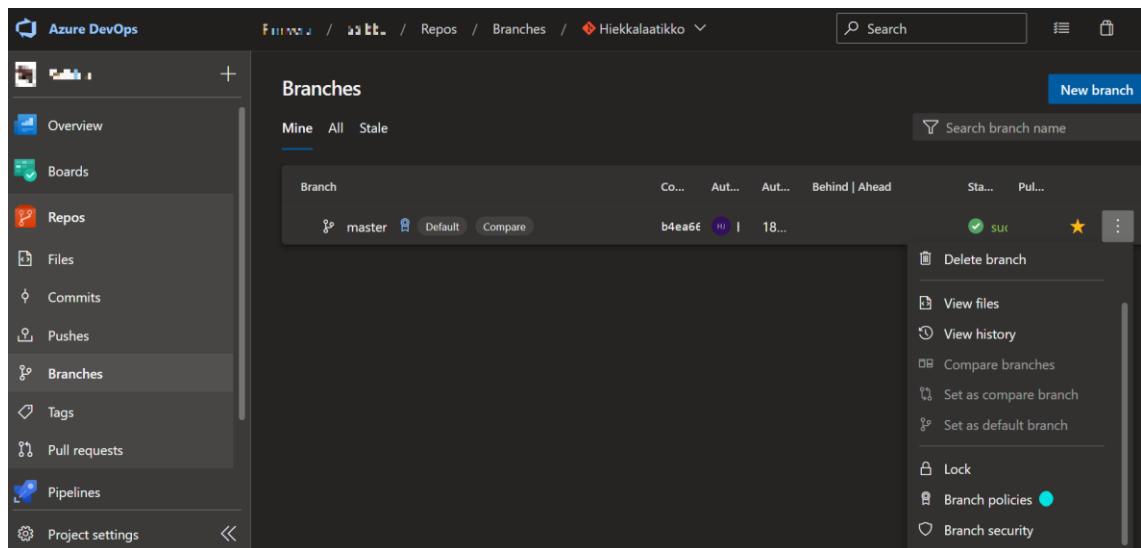
Koodiesimerkki 16. SonarQube-analyysin käynnistys.

SonarQubePublish-tehtävä koodiesimerkissä 17 julkaisee SonarQuben quality gaten Azureen. Syötteeksi on annettava pollingTimeoutSec, joka määrittää kuinka kauan tehtävä odottaa analyysin valmistumista.

```
- task: SonarQubePublish@4
  displayName: 'Publish SonarQube quality gate results'
  inputs:
    pollingTimeoutSec: '300'
```

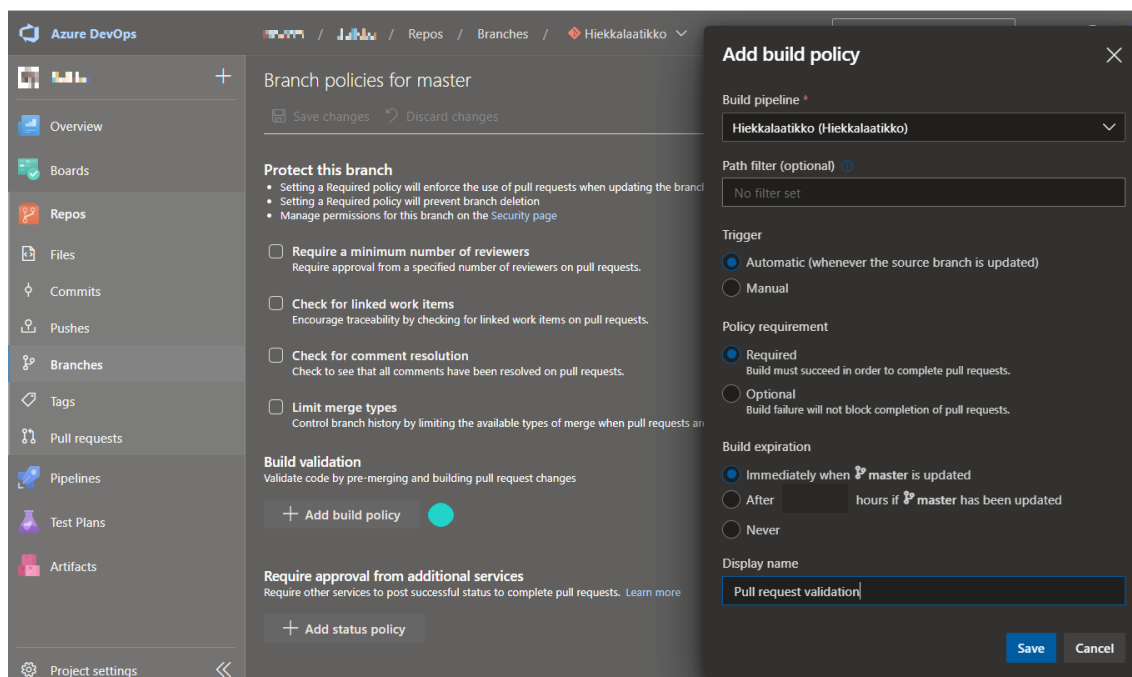
Koodiesimerkki 17. SonarQuben quality gaten julkaisu.

PR-validointiputki otetaan käyttöön määrittämällä se *branch policy*ksi päähaaraan.



Kuva 9. PR-validointiputki otetaan käyttöön siirtymällä Azure DevOps -alustalla Repos-valikon kautta Branches-sivulle ja avaamalla päähaaran asetukset, löytyy kohta *Branch policies*.



Kuvassa 9 näkyvästä asetusten kohdasta Build validation voidaan lisätä päähaaralle build policy. Policyyn määritetään, mitä putkea halutaan hyödyntää. Kuvan 10 esimerkissä lisää testiprojekti hiekkalaatikkoon saman nimisen PR-validointiputken. Koska putken YAML-tiedostossa oli triggeriksi määritelty *none*, määritellään se tässä automaattiseksi, eli aina kun haaraa päivitetään. Policyyn asetaan vaatimus, että koonnin on onnistuttava ennen päähaaraan yhdistämistä. Onnistunut koonti on voimassa niin kauan, kunnes päähaaraa päivitetään, jolloin koonti on suoritettava uudelleen. Näin voidaan varmistua, että uudet muutokset eivät aiheuta ongelmia muiden kehittäjien tekemien muutosten kanssa.



Kuva 10. Build policyn lisääminen päähaaraan.

Lisäksi branch policies -asetuksista on hyvä määrittää myös muita päähaaraa suojaavia asetuksia, kuten katselmoijien vähimmäismäärän, Azure Boardsiin määritettyjen työtehtävien linkittämisen seurannan helpottamiseksi sekä kommenttien selvityksen ennen PR:n hyväksyntää. Nämä kuvassa 11 näkyvät asetukset olivat projektissa jo aiemmin käytössä.

Branch policies for master

 Save changes
  Discard changes

Protect this branch

- Setting a Required policy will enforce the use of pull requests when updating the branch
- Setting a Required policy will prevent branch deletion
- Manage permissions for this branch on the [Security page](#)

☒ **Require a minimum number of reviewers**
Require approval from a specified number of reviewers on pull requests.

Minimum number of reviewers

☐ Allow requestors to approve their own changes
☐ Prohibit the most recent pusher from approving their own changes
☐ Allow completion even if some reviewers vote to wait or reject
☐ Reset code reviewer votes when there are new changes

☒ **Check for linked work items**
Encourage traceability by checking for linked work items on pull requests.

Policy requirement

☒ Required
Block pull requests from being completed unless they have at least one linked work item.
☐ Optional
Warn if there are no linked work items, but allow pull requests to be completed.

☒ **Check for comment resolution**
Check to see that all comments have been resolved on pull requests.

Policy requirement

☒ Required
Block pull requests from being completed while any comments are active.
☐ Optional
Warn if any comments are active, but allow pull requests to be completed.


☐ **Limit merge types**
Control branch history by limiting the available types of merge when pull requests are completed.

Kuva 11. Muut päähaaraa suojaavat asetukset.

Kuvassa 12 nähdään kaikki PR:lta vaaditut policyt. Varsinaiseen projektiin on lisätty toinenkin validointiputki, jonka tehtävä on ainoastaan koota projekti.

Policies

Required

- ✓ 2 reviewers approved
- ✓ Required reviewers have approved
- ✓ Work items linked
- ✓ All comments resolved
- ✓ ALL validation succeeded
- ✓  validation succeeded

Kuva 12. PR:n hyväksymisen edellytykset. PR:lle on asetettu kaksi tiimiläistä suorittamaan koodikatselmoinnin, ja he ovat hyväksyneet muutokset, PR:iin on linkitetty tehtävään liittyvä tarina (*work item*), kommentit on selvitetty sekä koon-tiputki, että PR-validointiputki ovat menneet läpi.

Lopputuloksena syntyi siis PR-validointiputki, joka suorittaa automaattisesti koonnin ja yksikkötestauksen sekä luo testikattavuusraportin. Kaikkiin ennalta määritettyihin toiminnallisuuksiin ei ylletty tämän opinnäytetyön aikana vanhojen kehysten tuomien yllättävien ongelmien sekä aikaa vievien tiimien välisten pyyntöjen odottelun vuoksi.

4.7 Jatkokehitysideoita

Koska kaikkia toiminnallisuuksia ei ehditty tämän opinnäytetyön puitteissa saada toimimaan, jäi jatkokehitykselle tarvetta. Tärkeimpänä voidaan pitää SonarQuben luoman quality gaten käyttöönottoa, joka edellyttää sellaisen testikattavuustyökalun löytämistä, joka toimisi myös vanhan koodin osalta. Projektiin tehtävien muutosten myötä koko projekti on tarkoitus vaihtaa SDK-tyyliprojektiksi (SDK-style project) seuraavan kahden vuoden aikana, jolloin on mahdollista hyödyntää tällä hetkellä toimimattomia raporttityökaluja.

Lisäksi selvitystöiden yhteydessä kävi ilmi, että Azure DevOps Service tarjoaa Serveriä enemmän toiminnallisuuksia, kuten sisäänrakennetun testikattavuusraportin luonnin agenteille ja WhiteSourcen integroinnin osaksi Azure Repos -palvelua, jolloin myös laajennus- ja lisäosaturvallisuus saataisiin katettua jo ennen päähaaraan yhdistämistä. Azure DevOps Serviceen siirtymisen kohdalla on kuitenkin otettava huomioon asiakkaan tietosuojavaatimukset, eli onko siirtyminen itse ylläpidetyltä, yksityiseltä palvelimelta Microsoftin tarjoamaan, Suomen rajojen ulkopuolella sijaitsevaan pilvipalveluun mahdollista. Tämä olisi kuitenkin hyvä jatkoselvityksen kohde.

5 Yhteenveto

Opinnäytetyön tavoitteena oli lisätä eräälle Samlinkin rahoitushallintaprojektin sovelluksen laatua parantavia jatkuvan integraation ratkaisuja. Projektia kehitetään Azure DevOps Server -ympäristössä. Projektin tuli lisätä PR-validointiputki, joka automatisoi koonnin, testauksen sekä SonarQube- ja WhiteSource-laatuanalyysit sekä 80 prosentin testikattavuusvaatimus. Näiden automaattisten toimintojen avulla vähennettiin kehittäjien subjektiivisen osaamisen ja muistin vaikutusta sovelluksen laatuun. Koonti, testaus ja laatuanalyysityökalut olivat käytössä lähtötilanteessa ainoastaan päähaaraa vasten ajettavassa putkessa, jolloin virheet havaittiin usein liian myöhään, ja aiheuttivat näin viivästyksiä kehitykselle, kun päähaaran korjausta jouduttiin odottamaan.

Lopputuloksena syntyi putki, joka sisältää koonnin, testauksen sekä SonarQube-analyysin. Selvityksistä kävi ilmi, että WhiteSource ei tue haara-analyysijä. Tämä toiminnallisuus olisi mahdollista saada käyttöön siirtymällä Azure DevOps Serveristä Microsoftin ylläpitämään Azure DevOps Serviceen. Tällöin WhiteSource on mahdollista integroida osaksi Azure Repos -palvelua. Testikattavuusraportti saatiin toimimaan ainoastaan testiprojektissa, joka on toteutettu uudemmalla .NET 6 -ohjelmistokehyksellä, kun taas varsinaisen projektin vanha koodi on toteutettu .NET Framework 4.7.2 -ohjelmistokehyksellä. Kyseinen ohjelmistokehys on sen verran vanha, ettei sitä tue useimmat avoimen lähdekoodin testikattavuustyökalut. Tämä on mahdollinen jatkoselvityksen aihe, mutta projektia ollaan siirtämässä kokonaisuudessaan vanhasta kehyksestä uudempaan, .NET 6:een seuraavien 1-2 vuoden kuluessa, jolloin testikattavuus on helpompi ottaa käyttöön. Koska testikattavuusraporttia ei saada luotua, ei SonarQube kykene tuottamaan quality gatea, jonka avulla 80 prosentin vähimmäisvaatimus testikattavuudelle voitaisiin asettaa.

Vaikka kaikkia toivottuja ominaisuuksia ei putkeen saatu, projektin on nyt olemassa alustava jatkuvan integraation ratkaisu, joka auttaa kehittäjiä ylläpitämään laadukasta koodipohjaa.

Lähteet

- 1 An Introduction to Continuous Integration (CI) and its Benefits. Verkkoaineisto. <<https://iclerisy.com/what-is-continuous-integration-ci/>> Luettu 22.1.2023.
- 2 Software Freedom Conservancy. 2020. Regarding Git and Branch Naming. Verkkoaineisto. <<https://sfconservancy.org/news/2020/jun/23/git-branchname/>> Luettu 24.1.2023.
- 3 Cooper, Matt. 2020. Azure Repos default branch name. Verkkoaineisto. <<https://devblogs.microsoft.com/devops/azure-repos-default-branch-name/>> Luettu 24.1.2023.
- 4 Häkkinen, Auvo. 2021. Ohjelmistojen laatu: SQuaRE. Luentomateriaali. Metropolia Ammattikorkeakoulu.
- 5 ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. Verkkoaineisto. <<https://www.iso.org/standard/35733.html>> Luettu 8.2.2023.
- 6 Fowler, Martin. Refactoring: Improving the Design of Existing Code.
- 7 Martin, Robert C. Clean code: A Handbook of Agile Software Craftmanship.
- 8 Technical debt standard: Estimating the cost of corrective maintenance. Verkkoaineisto. <<https://www.it-cisq.org/standards/technical-debt/>> Luettu 15.2.2023.
- 9 Fowler, Martin. TechnicalDebt. 21.5.2019. Verkkoaineisto. <<https://martinfowler.com/bliki/TechnicalDebt.html>> Luettu 15.2.2023.
- 10 Software Quality Standards – ISO 5055. Verkkoaineisto <<https://www.it-cisq.org/standards/code-quality-standards/>> Luettu 8.2.2023.
- 11 Nikolov, Rado. ISO 5055 standard explained. Is your software rock solid, efficient, and safe? 6.4.2021. Verkkoaineisto. <https://www.castsoftware.com/blog/iso-5055-standard-explained_is-your-software-rock-solid-efficient-and-safe> Luettu 15.2.2023.
- 12 Functional Vs Non-Functional Testing: From A to Z. 18.8.2020. Verkkoaineisto. <<https://u-tor.com/topic/functional-vs-non-functional>> Luettu 19.2.2023.

- 13 Häkkinen, Auvo. 2021. Ohjelmiston laatu: Testaus. Luentomateriaali. Metropolia Ammattikorkeakoulu.
- 14 What is software testing? Verkkoaineisto. <<https://www.ibm.com/topics/software-testing>> Luettu 9.2.2023.
- 15 Pittet, Sten. The different types of software testing. Verkkoaineisto. <<https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>> Luettu 9.2.2023.
- 16 Georgian, Stan. What Is End-To-End Testing and When Should You Use It? Verkkoaineisto. <<https://www.freecodecamp.org/news/end-to-end-testing-tutorial/>> Luettu 11.2.2023
- 17 Metric definitions. Verkkoaineisto. <<https://docs.sonarqube.org/latest/user-guide/metric-definitions/>> Luettu 18.2.2023.
- 18 Clean code for teams and enterprises with {SonarQube}. Verkkoaineisto. <<https://www.sonarsource.com/products/sonarqube/>> Luettu 18.2.2023.
- 19 Krasner, Herb. The Cost of Poor Software Quality in the US: A 2022 Report. Verkkoaineisto. <<https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/>> Luettu 18.2.2023.
- 20 OWASP Top Ten. Top 10 Web Application Security Risks. Verkkoaineisto. <<https://owasp.org/www-project-top-ten/>> Luettu 18.2.2023.
- 21 About the OWASP Foundation. Verkkoaineisto. <<https://owasp.org/about/>> Luettu 18.2.2023.
- 22 Mend. Mend Platform. Verkkoaineisto. <<https://www.mend.io/mend-platform/>> Luettu 21.2.2023.
- 23 What is Azure? Verkkoaineisto. <<https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/>> Luettu 1.2.2023.
- 24 What is cloud computing. Verkkoaineisto. <<https://learn.microsoft.com/en-us/training/modules/describe-cloud-compute/3-what-cloud-compute>>. Luettu 12.1.2023.
- 25 Describe the consumption-based model. Verkkoaineisto. <<https://learn.microsoft.com/en-us/training/modules/describe-cloud-compute/6-describe-consumption-based-model>> Luettu 20.2.2023.

- 26 Define cloud models. Verkkoaineisto. <<https://learn.microsoft.com/en-us/training/modules/describe-cloud-compute/5-define-cloud-models>>. Luettu 12.1.2023.
- 27 Describe Infrastructure as a Service. Verkkoaineisto. <<https://learn.microsoft.com/en-gb/training/modules/describe-cloud-service-types/2-describe-infrastructure-service>> Luettu 1.2.2023.
- 28 Describe Platform as a Service. Verkkoaineisto. <<https://learn.microsoft.com/en-gb/training/modules/describe-cloud-service-types/3-describe-platform-service>> Luettu. 1.2.2023.
- 29 Describe Software as a Service. Verkkoaineisto. <<https://learn.microsoft.com/en-gb/training/modules/describe-cloud-service-types/4-describe-software-service>> Luettu 1.2.2023.
- 30 Highsmith, Jim. History: The Agile Manifesto. Verkkoaineisto. <<https://agilemanifesto.org/history.html>> Luettu 11.2.2023.
- 31 Hall, Tom. DevOps vs. Agile. Verkkoaineisto <<https://www.atlassian.com/devops/what-is-devops/agile-vs-devops>> Luettu 4.2.2023.
- 32 Chai, Wesley. The Three Ways (The Phoenix Project). Verkkoaineisto. <<https://www.techtarget.com/whatis/definition/The-Three-Ways>> Luettu 4.2.2023.
- 33 Poppendick, Mary. Lean Software Development: The Backstory. Verkkoaineisto. <<https://www.leanessays.com/2015/06/lean-software-development-history.html>> Luettu 18.2.2023.
- 34 Courtemanche, Meredith. What is DevOps? The ultimate guide. Verkkoaineisto. <<https://www.techtarget.com/searchitoperations/definition/DevOps>> Luettu 24.01.2023.
- 35 Gunja, Saif. What is DevOps? Unpacking the purpose and importance of an IT cultural revolution. Verkkoaineisto. <<https://www.dynatrace.com/news/blog/what-is-devops/>>. Luettu 25.1.2023.
- 36 What is Azure DevOps? 2022. Verkkoaineisto. <<https://learn.microsoft.com/en-us/azure/devops/?view=azure-devops>> Luettu 12.1.2023.
- 37 What is Azure Pipelines? Verkkoaineisto. <<https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>> Luettu.30.1.2023.
- 38 Red Hat. 2021. What is YAML? Verkkoaineisto. <<https://www.redhat.com/en/topics/automation/what-is-yaml>> Luettu 30.1.2023.

- 39 Specify events that trigger pipelines. 10.2.2023. Verkkoaineisto.
<<https://learn.microsoft.com/en-us/azure/devops/pipelines/build/triggers?view=azure-devops>> Luettu 13.2.2023.
- 40 Add stages, dependencies, & conditions. 2022. Verkkoaineisto.
<<https://learn.microsoft.com/en-us/azure/devops/pipelines/process/stages?view=azure-devops&tabs=yaml>> Luettu 4.2.2023.
- 41 Specify jobs in your pipeline. 2022. Verkkoaineisto. <<https://learn.microsoft.com/en-us/azure/devops/pipelines/process/phases?view=azure-devops&tabs=yaml>> Luettu 4.2.2023.
- 42 Cross-platform targeting, Multi-targeting. Verkkoaineisto.
<<https://learn.microsoft.com/en-us/dotnet/standard/library-guidance/cross-platform-targeting>> Luettu 22.1.2023
- 43 Using Mend for Azure Repos. 13.2.2023. Verkkoaineisto.
<https://docs.mend.io/bundle/integrations/page/using_mend_for_azure_repos.html> Luettu 18.2.2023.