SeAMK
SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Henri Kotila

CI/CD Pipeline CODESYS-pohjaisessa ohjausjärjestelmässä

Opinnäytetyö
Kevät 2021
SeAMK tekniikka
Automaatiotekniikan tutkinto-ohjelma

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Tutkinto-ohjelma: Automaatiotekniikka

Suuntautumisvaihtoehto: Koneautomaatio

Tekijä: Henri Kotila

Työn nimi: CI/CD Pipeline CODESYS-pohjaisessa ohjausjärjestelmässä

Ohjaaja: Juha Yli-Hemminki

Vuosi: 2021 Sivumäärä: 41

Tämän opinnäytetyön tavoitteena oli analysoida modernin julkaisuputkitoteutuksen mahdollisuudet, ja toteuttaa asiakasprojekteja varten Pipeline, CODESYS-pohjaisten ohjelmistojulkaisujen kokoamista varten. Opinnäytetyön toimeksiantaja oli Epec Oy. Pipeline päätettiin toteuttaa Epecin asiakkaan Tana Oy:n Tana H-sarjan kaatopaikkajyrän ohjelmistoprojektille. Pipelinen päätehtävä oli automatisoida ohjelmistojulkaisun kokoamisprosessi. Toteutuksessa tuli ottaa huomioon, että pipelinea varten tehdyt työkalut olisi helppo ottaa käyttöön muihinkin asiakasprojekteihin.

Opinnäytetyössä käydään läpi, mitä tarkoitetaan jatkuvalla integraatiolla ja jatkuvalla julkaisulla/toimituksella. Lisäksi käsitellään CI/CD:n ennakkovaatimuksia, automaattisen testauksen merkitystä, iteratiivista ohjelmistokehitystä sekä Scrum- ja Kanban-toimintamalleja. Opinnäytetyössä tutustutaan siinä käytettyihin työkaluihin, joita olivat esimerkiksi Azure DevOps, CODESYS ja Python-ohjelmointikieli.

Opinnäytetyön tulokseksi saatiin Azure DevOpsin Pipeline, joka automaattisesti kääntää Tana H-projektin ohjelmistokoodin, kokoaa julkaisupaketin ja julkaisee sen Pipelinelle.

[1] Asiasanat: Ohjelmistojulkaisu, CODESYS, CI/CD, Pipeline

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree programme: Automation Engineering

Specialisation: Machine Automation

Author: Henri Kotila

Title of thesis: CI/CD Pipeline on CODESYS Based Control System

Supervisor: Juha Yli-Hemminki

Year: 2021 Number of pages: 41

The purpose and target of the thesis was to create a pipeline for a CODESYS based control system, which automatically builds the software code and compiles the release package. The thesis was commissioned by Epec Oy. The pipeline was decided to be made for Tana Oy's Tana H-series compactor software project. The main task of the Pipeline is to automate the software release process. Tools for Pipeline should be made so that they can easily be deployed for any Epec Oy customer project.

The thesis studied the benefits and potential of continuous integration and continuous delivery/deployment. The thesis also discussed the requirements of CI/CD, importance of automated testing, iterative software development, and Scrum and Kanban operating models. The tools introduced in the thesis, such as the Azure DevOps, CODESYS and Python programming language, were used in the implementation.

As the result of the thesis, Azure DevOps Pipeline was created for the Tana H customer project. This Pipeline automatically builds the software code, creates release package, and publishes it to the Pipeline.

[1] Keywords: Software release, CODESYS, CI/CD, Pipeline

SISÄLTÖ

Kuvaluettelo

Käytetyt termit ja lyhenteet

Azure DevOps	Ohjelmistokehitystä helpottavien menetelmien palvelu.
CI/CD Pipeline	Jatkuvan integroinnin/jatkuvan julkaisun putki, continuous integration/continuous delivery.
CODESYS	Epecillä käytössä oleva ohjelmointityökalu.
IEC 61131-3	Kansainvälinen standardi ohjelmoitaville logiikkaohjaimille.
OEM	Alkuperäinen laitevalmistaja, original equipment manufacturer.

1 JOHDANTO

1.1 Työn tausta

Ohjelmistosuunnittelussa jatkuva integrointi ja jatkuva toimitus/julkaisu on yleistynyt. Sitä kutsutaan CI/CD Pipelineksi. Tämän työn toimeksiantaja Epec Oy, haluaa olla kehityksessä mukana, ja ottaa käyttöön jatkuvan integroinnin ja jatkuvan julkaisun omiin asiakasprojekteihinsa. CI/CD Pipelinen tarkoituksena on kääntää ja tarkistaa automaattisesti eri projekteissa toteutettu ohjelmistokoodi ja rakentaa niistä tarvittaessa asiakkaalle julkaistava paketti. Näiden toimintojen automatisointi helpottaisi ja nopeuttaisi ohjelmistosuunnittelijoiden työtä. Kun koodia ei enää käännetä ja tarkisteta manuaalisesti, eikä julkaisupakettia rakenneta suunnittelijan työpöydällä, vältytään monilta inhimillisiltä virheiltä ja säästetään työaikaa. Manuaalisessa julkaisupaketin kokoamisessa tapahtuneet virheet huomataan asennettaessa julkaisupakettia testausympäristöön tai pahimmillaan vasta asiakkaan koneella. Mitä myöhemmin virhe huomataan, sitä kalliimmaksi se käy. Kun julkaisupaketti tehdään automaattisesti aina samalla tavalla, inhimillisiä virheitä ei tule. Jos jokin menee vikaan, siitä ilmoitetaan heti suunnittelijalle.

1.2 Työn tavoite

Tämän työn tavoitteena on analysoida modernin Pipeline-toteutuksen mahdollisuudet, ja toteuttaa Pipeline asiakasprojektien ohjelmistojulkaisujen kokoamista varten. Pipeline toteutetaan Epecin asiakkaalle, Tana Oy:n Tana H-sarjan kaatopaikkajyrän ohjelmistoprojektille. Työ päätettiin toteuttaa juuri tälle projektille, koska tämän työn tekijä oli aiemmin työskennellyt testaajana kyseisessä projektissa ja asiakkaalla oli kiinnostusta saada Pipeline käyttöön projektiinsa. Aikaisempi kokemus projektissa työskentelystä helpotti työn aloittamista. Pipeline on tarkoitus toteuttaa niin, että se voidaan tulevaisuudessa ottaa käyttöön helposti muihinkin asiakasprojekteihin. Pipelinen ohjelmistona tullaan käyttämään Azure DevOpsia (ADO). Azure DevOps ei tarjoa valmiita työkaluja CODESYS-ympäristössä toteutetuille ohjelmistoprojekteille, joten työkalut tullaan rakentamaan itse. CI/CD Pipelinen täytyy automaattisesti kääntää ja tarkistaa ohjelmistokoodi aina, kun versionhallintaan tulee uutta koodia, sekä käyttäjän niin halutessa, koota julkaisupaketti automaattisesti. Kaikkien vaiheiden tulokset raportoidaan Azure DevOpsiin.

1.3 Työn rakenne

Johdannossa käydään läpi opinnäytetyön taustaa, tavoitteita ja rakennetta. Lopuksi esitellään opinnäytetyön toimeksiantaja.

Teoriaosassa käsitellään jatkuvaa integrointia ja jatkuvaa toimitusta. Tämän jälkeen käydään läpi jatkuvan integroinnin historiaa sekä mitä ennakkovaatimuksia sillä on. Lopuksi perehdytään automaattisen testauksen merkitykseen, iteratiiviseen ohjelmistokehitysmalliin ja Scrum- ja Kanban-toimintamalleihin.

Kolmannessa luvussa esitellään ja käydään läpi työkalut, joita opinnäytetyön toteutuksessa on käytetty. Luvussa 4 käsitellään opinnäytetyön toteutus. Viidennessä luvussa käydään läpi työn tulokset ja pohditaan mahdollisia jatkokehitysmahdollisuuksia.

1.4 Epec Oy

Epec on Ponsse Oyj:n tytäryhtiö Seinäjoella. Se on erikoistunut liikkuvien työkoneiden ohjausjärjestelmiin. Veikko Rintamäki perusti yrityksen Seinäjoella vuonna 1978 nimellä E-P Elektroniikka. Yrityksen alkuvuosina liikeideana oli elektroniikkasuunnittelu ja valmistus sekä esim. elektroniikkakomponenttien ja tietokoneiden jälleenmyynti. Kun kansainvälinen toiminta alkoi kasvaa, vuonna 1989 yrityksen nimeksi vaihdettiin Epec. Hajautetun ja skaalautuvan ohjausjärjestelmän periaate kehitettiin 1990-luvun alussa, kun Epecillä suunniteltiin ensimmäisen sukupolven modulaarisia ohjausyksiköitä. Nämä tuotteet osoittautuivat menestykseksi ja saavuttivat nopeasti suosion suuren metsäkonevalmistajan sarjatuotantokoneissa. Seuraavien tuotesukupolvien aikana Epecin ohjausyksiköiden teknologia nousi uudelle tasolle. Moduulit kestivät paremmin hankalia olosuhteita niiden pöly- ja roisketiiviyden sekä tärinä- ja iskukestävyyden ansiosta. Vuonna 2004 Epecistä tuli osa metsäkoneyhtiö Ponsse Oyj:n konsernia. (Epec, [viitattu 5.2.2021].)

Nykyään Epec on liikkuvien työkoneiden ohjausjärjestelmien toimittaja, joka valmistaa ohjausyksiköitä ja näyttöjä, sähköajoneuvojärjestelmiä sekä avusteisia ja autonomisia järjestelmiä. Epecin monipuolinen kokemus perustuu pitkäaikaiseen yhteistyöhön johtavien kansainvälisten alkuperäisten laitevalmistajien (OEM) kanssa eri aloilla. (Epec Oy, [viitattu 6.2.2021].)

Epecin päätoimipiste ja tehdas sijaitsee Seinäjoella. Muut toimipisteet ovat Tampereella, Turussa ja Shanghaissa. Yhteensä työntekijöitä on yli 130. Epecin liikevaihto vuonna 2019 oli 24,7 M€. (Epec, [viitattu 5.2.2021].)

2 CI/CD (CONTINUOUS INTEGRATION/CONTINUOUS DELIVERY)

Tässä luvussa käydään läpi, mitä tarkoitetaan jatkuvalla integraatiolla (continuous integration) ja jatkuvalla julkaisulla/toimituksella (continuos delivery). Lisäksi perehdytään lyhyesti jatkuvan integraation historiaan ja käsitellään CI/CD:n ennakkovaatimuksia. Luvun lopussa kerrotaan automaattisen testauksen merkityksestä sekä iteratiivisesta ohjelmistokehityksestä.

2.1 Jatkuva integraatio (Continuous integration)

Jatkuva integraatio on ohjelmistotuotannon prosessi, jossa ohjelmistokoodi käännetään ja testataan automaattisesti. Automaattinen käännös ja testaus suoritetaan aina, kun ohjelmistokehitystiimin jäsen tuo muutoksia versionhallintaan. Jatkuva integraatio kannustaa ohjelmistokoodin kehittäjää tuomaan muutokset versionhallintaan vähintään päivittäin tai jokaisen suoritetun tehtävän jälkeen. (Guckenheimer 2017.) Jatkuvan integroinnin tarkoituksena on varmistaa, että ohjelmisto on jatkuvasti sellaisessa tilassa, että se on asennettavissa järjestelmään. Toisin sanoen koodikokoelmat kääntyvät ja koodin laadun voidaan olettaa olevan kohtuullisen hyvälaatuista. (Rossel 2017, 8.)

Jatkuvasta integraatiosta tuli hyödyllistä, kun koodia ei tuotu versionhallintaan ja käännetty suoritettavaan muotoon päiviin tai viikkoihin. Tämän tyyppinen kehittäminen altisti ohjelmistokoodin monille virheille. Tästä syystä koodin jatkuva automaattinen kääntäminen ja testaaminen on hyödyllistä. Ohjelmistokoodin virheet huomataan paljon aikaisemmin, koska koodi käännetään ja testataan joka kerta kun versionhallintaan tuodaan muutoksia. Virheiden tullessa ilmi aikaisemmin, niiden korjaaminen on paljon halvempaa. (Guckenheimer 2017.)

2.2 Jatkuva julkaisu/toimitus (Continuous delivery)

Jatkuva julkaisu on ohjelmistotuotantoprosessi, jonka tarkoituksena on pitää ohjelmistokoodi aina sellaisessa tilassa, että sen voi toimittaa asiakkaalle milloin vain. Jatkuvan julkaisun ansiosta ohjelmisto on julkaisukelpoinen koko sen elinkaaren ajan. Kuka tahansa tiimin jäsen voi saada nopeaa ja automatisoitua palautetta järjestelmiensä toimitusvalmiudesta, kun joku tekee niihin muutoksia. Tarvittaessa voidaan julkaista ohjelmiston mikä tahansa versio mihin tahansa ympäristöön. (Fowler 2013.) Jatkuva integraatio on edellytys onnistuneelle jatkuvalle julkaisulle/toimitukselle (Rossel 2017, 18).

Jatkuvan julkaisun vaiheet ovat:

- koodin muutokset ovat pieniä ja ne siirretään versionhallintaan usein
- ongelmien havaitsemiseksi ohjelmistokoodi käännetään ja sille suoritetaan automaattisia testejä
- lopuksi ohjelmisto ladataan simulaattoriin tai tuotannon kaltaiseen ympäristöön ja varmistetaan, että ohjelmisto toimii tuotannossa (Fowler 2013).

Jatkuva julkaisu helpottaa julkaisuprosessia niin paljon, että julkaisuja voi tehdä paljon useammin. Ilman jatkuvaa julkaisua julkaisun tekeminen vie vähintään tunnin kehittäjän ajasta. Manuaalisessa julkaisupaketin kokoamisessa on huomattavia epäonnistumisriskejä. Kehittäjä haluaa lykätä julkaisua niin kauan kuin mahdollista. Tämä johtaa siihen, että julkaisuun ehtii kerääntyä monia muutoksia. Koska muutoksia on paljon, se lisää riskiä siihen, että muutoksissa on jokin virhe. Useiden muutoksien seasta on vaikea löytää virheen aiheuttajaa. Julkaisuprosessin helpottaminen automatisoinnilla kannustaa kehittäjää siihen, että pienempiä muutoksia julkaistaan useammin. Kun julkaisujen muutokset ovat pieniä, vikatilanteessa on helpompi palata takaisin aikaisempaan julkaisuun. (Rossel 2017, 19.)

2.3 Historia

Continuous integration -termiä on käytetty 1990-luvun puolivälistä alkaen. Vuonna 1998 keksittiin ”Extreme Programming” (XP) ohjelmistokehitystapa. XP on ketterä ohjelmistokehitystapa, jonka tavoitteena on tuottaa laadukkaampia ohjelmistoja ja parempaa työn laatua kehitystiimille. Jatkuva integrointi on mukana tässä ohjelmistokehitystavassa. Vuonna 2001 ”Cruise Control” oli ensimmäinen avoimen lähdekoodin jatkuvan integraation työkalu mukautetun koontiprosessin luomiseen. (Devopedia 2020.)

2.4 Ennakkovaatimukset

Jotta jatkuva integrointi on mahdollista ottaa käyttöön ohjelmistokehitysprojektiin, sillä on muutamia vaatimuksia. Ensimmäinen niistä on versionhallintajärjestelmä. Versionhallintajärjestelmän tarkoitus on hallita ohjelmistoprojektin tuotosta, kuten lähdekoodeja. Versionhallinta pitää muistissaan jokaisen tiedoston aiemmat muutokset, jotta niihin on myöhemmin helppo palata. Ilman versionhallintaa, kehittäjän täytyy säilyttää koodista

useita kopioita omalla tietokoneellaan. Vaarana on, että vahingossa poistaa tai muuttaa tiedostoja väärästä kopiosta, ja hävittää työn. (Outlaw 2017.)

Toinen vaatimus on automatisoitu käännösprosessi (build). Käännösprosessi kääntää lähdekoodit sellaiseen muotoon, että ne voidaan suorittaa. Käännös on tehtävä aina, kun kehittäjä tekee versionhallintaan muutoksia. Kun käännöksiä tehdään usein, on helpompi pysyä perillä siitä, mikä muutos/lisäys aiheuttaa virheellisen koodikäännöksen. (Fowler 2006.)

Kolmas vaatimus on, että kehittäjätiimi sitoutuu toimimaan vaaditulla työskentelymallilla. Muutamia tärkeitä toimintatapoja on lueteltu seuraavassa:

- Koodin muutokset viedään versionhallintaan riittävä usein
- Versionhallintaan viedään vain ehjä koodi
- Käännösprosessissa ilmenneet virheet korjataan heti
- Testausympäristö pidetään kunnossa
- Käyttöön otetaan automaattisia testejä
- Käännösprosessi pidetään lyhyenä. (Fowler 2006.)

On tärkeää, että tiimi suhtautuu automatisoituun käännösprosessiin vakavasti. Käännösprosessi on hyvä pitää lyhyenä, jotta palautetta saadaan välittömästi, jos käännös epäonnistuu. Kun ohjelmistokehittäjä tuo versionhallintaan koodia, joka tekee käännösprosessista epäonnistuneen, käännösprosessin korjauksen tulisi tulla ensisijaista. Tärkeintä on pitää mielessä, että käännösprosessin epäonnistuessa on mahdotonta saada toimivaa koodia, joka läpäisee kaikki testit ja laatuvaatimukset (Rossel 2017, 18).

2.5 Automaattisen testauksen merkitys

Automatisoitu ohjelmistojen testaus tarkoittaa automaattisten menetelmien hyödyntämistä ohjelmistojen testauksessa koko ohjelmiston elinkaaren ajan. Automattisen testauksen tavoitteena on tarjota nopeampaa ja tehokkaampaa tuotettavuutta. Ammattitaitoisesti toteutettu ja potentiaalisesti käytetty automaatio on erittäin hyödyllistä ja vaikuttaa siihen, kuinka suuria ja monimutkaisia ohjelmistojärjestelmiä testataan ja toimitetaan. (Innovative Defense Technologies, [viitattu 24.2.2021].)

Seuraavassa luettelossa on automaattisen testauksen hyötyjä manuaaliseen testaukseen verrattuna:

- Laadukkaammat ohjelmistot: Toistuvat, johdonmukaiset ja perusteelliset testit tukevat laadukkaiden ohjelmistojen toimittamista.
- Parempi dokumentaatio: Automatisoidut testit tuottavat helposti kattavia raportteja testien hyväksymis- ja hylkäystuloksista.
- Pienemmät testausajat sekä työvoimaresurssit: Automaattiset testit suoritetaan huomattavasti nopeammin kuin manuaaliset testit. Testaajalle jää paljon aikaa muihin tehtäviin.
- Kustannussäästöt: Automaatio voi vähentää ohjelmistojen tuottamiseen liittyviä kustannuksia. Säästöjä syntyy lyhentyneestä testausajasta ja työvoimasta, mutta myös alhaisimmista elinkaarikustannuksista parantuneen ohjelmistojen laadun ja dokumentoinnin ansiosta. (Innovative Defense Technologies, [viitattu 24.2.2021].)

2.6 Iteratiivinen ohjelmistokehitys

Tunnetuin ohjelmistokehitysmalli on vesiputousmalli. Vesiputousmallissa ohjelmistokehitysprosessi etenee vaihe vaiheelta eteenpäin, kuin vesiputouksessa. Vesiputousmallin jäykkyys on luonut tarpeen uusille kehitysmalleille, jotka tuottaisivat nopeampia tuloksia ilman tarkkoja ja yksityiskohtaisia etukäteen saatavia tietoja. Vesiputousmallissa tuote on suunniteltu siten, että ohjelmistosta saadaan toimiva versio yhden jakson lopussa. Toisin kuin vesiputousmallissa, iteratiivisessa kehityksessä tuote on jaettu pieniin osiin, joita kehitetään sarjana peräkkäisiä syklejä. Tämän avulla kehitysprosessia ja tuotearkkitehtuuria ei tarvitse määritellä tarkasti alusta alkaen. (Muffatto 2006, 76 - 77.)

Iteratiivisen kehityksen jokaisen syklin tuotoksena on osittain käyttökelpoinen tuote, jota voidaan testata. Testauksesta saadun palautteen ansiosta tuotetta voidaan kehittää ja parantaa. Jokainen tuotteen uusi versio parantaa edellisissä sykleissä kehitettyjä toimintoja. Kehitysten tulokset ovat saatavissa aikaisemmin ja palautetta voidaan käyttää vaatimusten muuttamiseen ja parantamiseen koko prosessin ajan. Kun kehitysprosessi on iteratiivinen, se luo mahdollisuuden jatkuvan julkaisun/toimituksen (CD) käyttöönottoon. Yhdessä iteratiivinen kehitystapa ja jatkuva julkaisu/toimitus antavat tiimille mahdollisuuden julkaista kaikki ominaisuudet, kun ne ovat valmiita, sen sijaan että ne ryhmitellään yhteen lopullista julkaisua varten. Tämä luo tehokkaamman kehitysprosessin. (Muffatto 2006, 76 - 77.)

Iteratiivisen kehitystavan haittana voi olla se, että projektit voivat laajeta ja muuttua merkittävästi alkuperäisestä suunnitelmasta. Iteratiivinen kehitys ja jatkuva palaute voivat saada kehittäjiä suunnittelemaan ja ottamaan käyttöön loputtomasti uusia ominaisuuksia. Iteratiivinen kehitys soveltuu parhaiten innovatiivisille projekteille. Näissä projekteissa vaatimuksia ei tunneta tarkasti, ja on mahdotonta ennakoida projektin kehitystä. (Muffatto 2006, 76 - 77.)

2.7 Ketterät toimintamallit – Scrum ja Kanban

Scrum ja Kanban ovat tunnettuja ketteriä toimintamalleja. Näiden molempien tavoitteena on tuottaa hyvää laatua ja tehokasta työskentelyä.

Scrum käyttää lyhyitä iteraatioita, joita kutsutaan sprinteiksi. Jokaisen sprintin tavoitteena on suorittaa sprintille suunnitellut tehtävät ja rakentaa niistä testattu ja toimiva osa järjestelmään. Sprintti alkaa suunnittelukokouksella, jossa tiimi katsoo läpi sprinttiin kuuluvat työt. Tiimin jäsenet antavat aika-arviot heille kuuluvista töistä. Tämän avulla päästään selville siitä, ettei tiimille pakoteta liikaa tehtävää, ja kaikki aloitetut asiat saadaan valmiiksi. (Robson 2013, 25 - 27.)

Kanbanissa ei ole sprinttejä, vaan työtä tehdään jatkuvasti, yksi asia kerrallaan. Kun yksi tehtävä saadaan valmiiksi, se siirretään Kanban-taulussa seuraavaan kohtaan. Työ otetaan prosessiin prioriteetin perusteella heti, kun resursseja on käytettävissä. Kanban on kuin liukuhihna, jossa työtehtäviä ei voi puskea eteenpäin, jos hihna on tukossa. Työn virtaus on tärkeää pitää hallinnassa. Pitää välttää tilanteita, jossa tehtäviä on liikaa yhtä aikaa meneillään. Kanban-toimintamalli on joustava. Kanban sallii sen, että kiireelliset tehtävät voidaan ottaa työjonon kärkeen, ja ottaa nopeasti työn alle. (Robson 2013, 27 - 31.)

3 KÄYTETYT TYÖKALUT

3.1 Azure DevOps

Azure DevOps on palvelu, joka tarjoaa ohjelmistokehitystä helpottavia menetelmiä esimerkiksi toiminnanohjaukseen, versionhallintaan ja CI/CD-putkien kehitykseen. DevOps-termi tulee sanoista development (kehitys) ja operations (palvelut). (Microsoft 2021.) Esimerkiksi Donovan Brown (2015) määrittelee DevOpsin seuraavasti: ”DevOps on ihmisten, prosessien ja tuotteiden yhdistelmä, joka mahdollistaa arvon jatkuvan toimittamisen loppukäyttäjillemme.” Seuraavissa alaluvuissa käydään läpi Azure DevOpsin tarjoamat palvelut.

3.1.1 Azure Artifacts

Azure Artifacts on DevOpsin sisäinen paketinhallintajärjestelmä, jonka avulla voidaan luoda ja jakaa esimerkiksi Maven-, npm-, NuGet- ja Python-paketteja julkisista ja yksityisistä lähteistä (Microsoft, [viitattu 10.2.2021]). Paketit tallentuvat järjestelmään ja niitä voidaan käyttää esimerkiksi Azure Pipelines CI/CD -työkalussa. Azure Pipelines -palvelua voidaan käyttää julkaisemaan ja tallentamaan paketteja, sekä integroimaan tiedostoja CI/CD-putken vaiheiden välillä. Sitten paketteja voidaan lisätä, luoda, testata tai julkaista. (Microsoft 2020a.)

3.1.2 Azure Boards

Azure Boards on palvelu ohjelmistoprojektien työn hallintaan. Se tarjoaa runsaasti ominaisuuksia, kuten Scrumin ja Kanbanin tuen, sekä integroidun raportoinnin. (Microsoft Azure 2018.) Tällä voidaan hallinnoida ja seurata projektin tehtäviä, virheitä ja ominaisuuksia, suunnitella sprinttejä ja muokata näkymiä, jotka kuvaavat projektin tilaa. (Microsoft 2020b.)

3.1.3 Azure Pipelines

Azure Pipeline -palvelu kääntää ja testaa automaattisesti projektin koodin. Se toimii melkein millä tahansa ohjelmointikielellä tai projektityypillä. Azure Pipeline yhdistää jatkuvan integroinnin (CI) ja jatkuvan julkaisun/toimituksen (CD). Pipeline kääntää ja testaa koodin aina, kun versionhallintaan tulee muutoksia ja toimittaa julkaisupaketin mihin tahansa kohteeseen. Jatkuva integrointi auttaa löytämään virheet aikaisemmin, jotta ne olisi mahdollisimman halpa

korjata. Jatkuva julkaisu/toimitus on prosessi, jossa koodi käännetään, testataan ja julkaistaan. (Microsoft 2019a.)

3.1.4 Azure Repos

Azure Repos on versionhallintatyökalu, jota voidaan käyttää koodin hallintaan. Versionhallinnan avulla voidaan seurata koodissa tehtyjä muutoksia. Kun koodia muokataan, versionhallintajärjestelmä tallentaa muistiin tilannekuvan tiedostoista. Aiemmat tiedostojen versiot tallentuvat versionhallintaan pysyvästi, jotta niihin voidaan tarvittaessa palata myöhemmin. Vaikka ohjelmistokehitystiimi olisi pieni, versionhallinta auttaa pysymään järjestyksessä, kun virheitä korjataan ja uusia ominaisuuksia kehitetään. Versionhallinta tallentaa historian, jotta on helppoa tarkistaa ja palata projektin mihin tahansa versioon. (Microsoft 2020c.)

3.1.5 Azure Test Plans

Laatu on tärkeä osa ohjelmistojärjestelmää. Jotta laatu voidaan varmistaa, täytyy ohjelmistojärjestelmät testata. Azure DevOpsin Test Plans tarjoaa työkaluja, joita voidaan käyttää ohjelmistoprojektin testaamiseen sekä laadun ja yhteistyön edistämiseen. Test Plansilla voidaan suunnitella ja suorittaa manuaalisia testejä, käyttäjän hyväksyntätestejä ja kokeellisia testejä. Lisäksi Test Plansilla voidaan kerätä palautetta suoritetuista testeistä. (Microsoft 2019b.)

3.2 CODESYS

CODESYS on johtava, valmistajasta riippumaton IEC 61131-3 -standardin mukainen ohjelmistoympäristö, ohjausjärjestelmien suunnitteluun (CODESYS, [viitattu 13.2.2021]). Vuonna 1994 Dieter Hess ja Manfred Werner perustivat 3S-Smart Software Solutions GmbH:n, jonka nimeksi vaihtui CODESYS GmbH kesäkuussa 2020. CODESYS BmbH on osa CODESYS-konsernia. Yritys työllistää yli 170 henkilöä, joita työskentelee pääkonttorissa Saksassa ja tytäryhtiöissä Kiinassa, Italiassa ja Yhdysvalloissa. Dieter Hess ja Manfred Werner ovat CODESYS-konsernin teknisen ja taloudellisen kehityksen pääjohtajia. (CODESYS, [viitattu 14.2.2021].)

CODESYS-ohjelmistoympäristöä käytetään kaikilla automaatioteollisuuden aloilla, kaikenlaisissa älykkäissä sovelluksissa, tehtaista koneenrakennukseen. Yli 400 laitevalmistajaa on integroinut CODESYS-ympäristön laitteisiinsa. CODESYS-ympäristöllä on kymmeniä tuhansia käyttäjiä ympäri maailmaa päivittäin. (CODESYS, [viitattu 14.2.2021].)

3.3 Python-ohjelmointikieli

Python on tulkattava ja vuorovaikutteinen olio-ohjelmointikieli. Tulkattava ohjelmointikieli tarkoittaa sitä, että Python-ohjelman suorittamiseksi tarvitaan erillinen tulkki (ohjelma), joka lukee Python-koodin ja toteuttaa sen käskyt. Python-ohjelmointikieli on suunniteltu yksinkertaiseksi ja helppokäyttöiseksi. Se sopii hyvin aloittelevalle kuin kokeneelle ohjelmoijalle. Pythonissa on vain vähän yksityiskohtia, joita ohjelmoijan täytyy opetella. Yleisien ohjelmointiongelmien ratkaisemiseksi tarvittavia valmiita työkaluja Pythonissa on runsaasti. (Korpela 2001.) Python Package Index (PyPi) tarjoaa tuhansia kolmannen osapuolen moduuleja, kuten Pythonin-standardikirjaston sekä yhteisön tuottamia moduuleita. Se on siis Python-ohjelmointikielen ohjelmistovarasto. Sieltä löytyy tietyllä toiminnallisuudella varustettuja kirjastoja, jotka tuovat lisää toiminnallisuuksia ohjelmaan. (Python, [viitattu 15.2.2021].)

Python on kehitetty Open Source Initiativen (OSI) hyväksymällä avoimen lähdekoodin lisenssillä, mikä tekee siitä vapaasti käytettävän ja jaeltavan jopa kaupalliseen käyttöön. Pythonin lisenssiä hallinnoi Python Software Foundation. (Python, [viitattu 15.2.2021].) Python Software Foundation on voittoa tavoittelematon yhtiö, joka hallinnoi lisensointia Python-versioille ja rahoittaa Pythoniin liittyvää kehitystä (Python, [viitattu 16.2.2021]).

3.4 Agentti

Azure Pipelinen agentti on ”työntekijä”, joka suorittaa suurimman osan Pipelinen töistä yksi kerrallaan. Agenttia tarvitaan, kun halutaan esimerkiksi kääntää ohjelmistokoodi tai rakentaa julkaisupaketti. Pipeline tarvitsee vähintään yhden agentin. Kun työntekijämäärä ja ohjelmistot kasvavat, tarvitaan useampi agentti. (Microsoft 2020d.)

Jos Pipeline on Azure Pipelines -palvelussa, on kätevä vaihtoehto suorittaa työt Microsoftin isännöimän agentin avulla. Microsoftin isännöimien agenttien avulla ylläpito ja päivitykset

hoidetaan käyttäjän puolesta. Aina kun Pipelinea suoritetaan, uusi virtuaalikone luodaan käyttäjälle. Virtuaalikone hävitetään yhden käyttökerran jälkeen. (Microsoft 2020d.)

Toinen vaihtoehto on itse isännöity agentti. Se on agentti, jonka ominaisuuksia voi itse määritellä ja hallinnoida. Itse isännöityjä agentteja on mahdollista käyttää Azure Pipelinesissa tai Team Foundation Serverissä (TFS). Itse isännöidyt agentit antavat paremmat mahdollisuudet tarvittavien ohjelmien asennuksiin ja hallintaan. Jos Pipelinessa on useampi agentti, jokaisella itse isännöidyllä agentilla on joukko ominaisuuksia, jotka osoittavat mitä se voi tehdä. Valmiudet ovat nimi-arvo-pareja, jotka agenttiohjelmisto havaitsee automaattisesti ja valitsee oikean agentin suorittamaan halutun työn. (Microsoft 2020d.)

4 TOTEUTUS

4.1 Taustatutkimus

Työn toteuttaminen alkoi aiheeseen tutustumisella ja opiskelulla. Tietoa etsittiin CI/CD-putkesta, Azure DevOpsista sekä muista aiheeseen liittyvistä asioista. Ponsse järjesti esittelyn heidän CI/CD-prosesseistaan ja työkaluistaan. Tuosta esityksestä selvisi, kuinka laajasta kokonaisuudesta onkaan kyse. Tässä vaiheessa päätettiin, että opinnäytetyö täytyy rajata järkevästi ettei työ kasva liian suureksi. Myös Epecin tuotekehitys esitteli omia Pipeline-prosessejaan.

Työ toteutetaan Tana Oy:n Tana H-sarjan kaatopaikkajyrän ohjelmistoprojektiin (Kuva 1).

Kuva 1. Tana H-sarjan kaatopaikkajyrä (Tana, [viitattu 20.2.2021]).

Projektiryhmän kanssa käytiin läpi, kuinka tällä hetkellä manuaalisesti käännetään ohjausyksiköiden ohjelmistokoodit ja kootaan julkaisupaketti. Tässä kirjattiin muistiin kaikki julkaisupaketin kokoamisen vaiheet sekä tiedostot, jotka pitää olla paketin mukana. Julkaisupaketin kokoamisessa on paljon tiedostojen kopiointia ja liittämistä. Nuo toiminnot voidaan helposti automatisoida.

4.2 Moduulien käännökset komentoriviltä

Koska Azure DevOps ei tue suoraan CODESYS-käännöksiä, päätettiin toteuttaa oma Python-pohjainen työkalu käännösten sekä muiden ohjelmistojulkaisupaketin vaatimien vaiheiden toteuttamiseen. Tätä työkalua tullaan käyttämään ADO Pipelinen kautta. Ensimmäinen työvaihe oli saada käännettyä ohjausyksiköiden (moduulien) ja näytön sovelluskoodit suoraan komentoriviltä (Command prompt). Tässä projektissa ohjausyksiköinä toimii Epecin 3724 Control Unit (Kuva 2) ja 5050 Control Unit (Kuva 3). Näyttönä toimii Epecin 7 tuuman 6107 Display Unit (Kuva 4).

Kuva 2. Epec 3724 Control Unit (Epec, [viitattu 20.2.2021]).

Kuva 3. Epec 5050 Control Unit (Epec, [viitattu 21.2.2021]).

Kuva 4. Epec 6107 Display Unit (Epec, [viitattu 22.2.2021]).

Koska ohjausyksiköt 5050 ja 6107 käyttävät CODESYS-versiota 3.5 ja ohjausyksikkö 3724 versiota 2.3, komentorivikomennot käännösten tekemiseen tulevat olemaan erilaisia. Epecin toisessa asiakasprojektissa käännöksiä komentoriviltä oli jo tehty, joten sieltä saatiin hyviä vinkkejä. 3724-moduulin käännös tehtiin kuvan 5 tavalla.

build_23_module.bat

```
1  call "C:\Program Files (x86)\3S Software\CoDeSys V2.3\Codesys.exe" C:\path_to_project\module1.pro %Hidestatus%  /cmd "script1.cmd"
```

Kuva 5. CODESYS 2.3 -komentorivikomento.

Komentorivikomennossa kutsuttiin CODESYS-ohjelman 2.3 versiota, sekä annettiin käännettävän moduulin projektin polku. Lopussa viitattiin script.cmd-tiedostoon, jossa annettiin CODESYS-ympäristölle oikeat käännöskomennot (Kuva 6).

build_23_module.bat | script1.cmd

```
1  onerror continue
2  query off ok
3  project clean
4  project build
5  delay 3000
6  online bootproject
7  delay 1000
8  file save
9  delay 3000
```

Kuva 6. CODESYS 2.3 -käännöskomennot.

Ohjausyksiköiden 5050 ja 6107 käännökset toteutettiin kuvan 7 tavalla. Komentorivikomennossa kutsuttiin CODESYS-ohjelmaa, kerrottiin CODESYS-versio ja lopussa ajettiin Python-skripti ”build_35.py”.

build_35_module.bat

```
start /wait CODESYS.exe --profile="CODESYS V3.5 SP13" --runscript="build_35.py"
```

Kuva 7. CODESYS 3.5 -komentorivikomento.

Python-skripti ”build_35.py” (Kuva 8) avaa CODESYS-projektin ja tekee käännöksen. Käännöksen tuloksena on tässä tapauksessa ”Application.app” -binääritiedosto.

build_35.py

```
from __future__ import print_function

import os


# Open project
proj = projects.open(project_path)

app = proj.active_application

# Create boot application
app.create_boot_application('Application.app', update_compile_info=True, write_visu_files=True)
system.exit()
```

Kuva 8. CODESYS 3.5 -käännöskomennot.

4.3 YAML-tiedoston määrittely

Kun kaikki käännökset onnistuttiin tekemään komentoriviltä, seuraava vaihe oli saada kaikki konfiguroitava tieto yhteen paikkaan. Eli esimerkiksi moduulien CODESYS-projektien polut, käytettävät CODESYS-versiot, sekä julkaisupakettiin tulevat tiedostot ja polut. Tiedot päädyttiin määrittelemään YAML-tiedostoon. YAML on helppokäyttöinen tietojen merkintäkieli, joka sopii kaikille ohjelmointikielille. YAML on lyhenne sanoista ”YAML Aint’t Markup Language”. (Yaml [viitattu 2.3.2021].) YAML-tiedosto avataan Python-skriptissä PyYAML-kirjaston avulla, näin saadaan kaikki tarvittava tieto käyttöön. Kun kaikki tiedot ovat yhdessä paikassa, saadaan määriteltyä työ pois skriptien sisältä. Kun tätä Python-työkalua otetaan

käyttöön tulevaisuudessa toisessa projektissa, riittää että määritellään projektille oma YAML-tiedosto (Kuva 9).

```yaml
---
# Package definitions for release package
version: 0.01.00
package_name: Package_files_
release_package_name: ReleasePackage_V

codesys23_path: C:\\Program Files (x86)\\3S Software\\CoDeSys V2.3\\Codesys.exe
codesys35_profile: CODESYS V3.5 SP13
build_35_module_path: D:\\builds\\1\\s\\build_35_module.py
build_23_module_path: D:\\builds\\1\\s\\build_23_module.cmd

machine_type_params_excel_path: D:\\builds\\1\\s\\path_to_file\\file.xls
machine_type_params_macro: PressButton
language_excel_path: D:\\builds\\1\\s\\path_to_file\\file1.xls
language_macro: PressButtons
zip_destination: D:\\builds\\1\\s\\work_folder\\release
work_folder: D:\\builds\\1\\s\\work_folder\\package_files
source_code_folder: D:\\builds\\1\\s\\path_to_source

# List of modules which will be build for the release
Modules:
  - Module_name: Module1
    output: Module1.bin
    path: D:\\builds\\1\\s\\path_to_project\\Module1.pro
    codesys_version: 2.3
  - Module_name: Module2
    output: Module2.bin
    path: D:\\builds\\1\\s\\path_to_project\\Module2.project
    codesys_version: 3.5
  - Module_name: Module3
    output: Module3.app
    path: D:\\builds\\1\\s\\path_to_project\\Module3.project
    codesys_version: 3.5

# Dependencies to be included in release package
Dependencies:
  - Folder1:
    source_location: D:\\builds\\1\\s\\path_to_folder
    destination_location: D:\\builds\\1\\s\\work_folder\\package_files
    exclude:
      - file.txt
  - Folder2:
    source_location: D:\\builds\\1\\s\\path_to_folder\\Folder2
    destination_location: D:\\builds\\1\\s\\work_folder\\package_files\\Folder2
  - File1:
    source_location: D:\\builds\\1\\s\\path_to_file\\File1
    destination_location: D:\\builds\\1\\s\\work_folder\\package_files
```

Kuva 9. Esimerkki YAML-tiedostosta.

4.4 Python-ohjelmointi

YAML-tiedoston lukemiseen Pythonilla käytettiin PyYAML-kirjastoa. Kuvassa 10 on esimerkki YAML-tiedostossa olevien tietojen hakemisesta. Tiedot määritellään muuttujiksi, joita käytetään myöhemmin.

```
with open(build_config) as f:
    data = yaml.load(f, Loader=yaml.FullLoader)
    codesys23_path = data['codesys23_path']
    codesys35_profile = data['codesys35_profile']
    build_35_path = data['build_35_module_path']
    build_23_path = data['build_23_module_path']
    module_list = data['Modules']
```

Kuva 10. Tietojen haku YAML-tiedostosta.

YAML-tiedostosta saatiin tietoon, mitä moduuleja kyseisellä projektilla on ja mitä CODESYS-versiota ne käyttävät. Python-koodissa käydään jokainen moduuli yksitellen läpi ja tehdään sille oikea käännöskomento. Pythonilla komentorivikomennot toteutettiin subprocess-kirjastolla. Kuvassa 11 on esimerkki, kuinka moduulilista käydään läpi for-loopilla, tarkistetaan jokaisen moduulin CODESYS-versio ja tehdään käännöskomento sen mukaan. YAML-tiedostosta haetaan CODESYS-profiili (esim. SP13), CODESYS-projektin polku ja käännöksestä ulos tulevan binääritiedoston nimi (output). CODESYS 3.5 -version käännöskomennossa ajetaan Python-skripti, jonka polku haetaan YAML-tiedostosta (build_35_path). Skriptissä on käännöskomennot, jotka on esitelty aiemmin kuvassa 8.

```
for module in module_list:
    project_path = module['path']
    module_name = module['Module_name']
    output = module['output']
    print("Building module: {}".format(module_name))

    if module['codesys_version'] == 3.5:
        build_35_module = f'start /wait CODESYS.exe --noUI ' \
                          f'--profile=\"{codesys35_profile}\" ' \
                          f'--runscript="{build_35_path}" ' \
                          f'--scriptargs:"{project_path} {output}"'
        proc = subprocess.Popen(build_35_module, shell=True)
```

Kuva 11. Moduulien läpikäynti ja CODESYS 3.5 -komentorivikomento.

Jos moduulin CODESYS-versio on 2.3, komentorivikomento tehdään kuvan 12 mukaan. YAML-tiedostosta haetaan CODESYS-ohjelman polku, projektin polku sekä moduulin nimi, joka lisätään käännöksestä ulos tulevan lokitiedoston nimeen. Lokitiedostosta voidaan tarkastella käännöksen vaiheita ja virhetilanteessa etsiä ongelman aiheuttajaa. Komennossa kutsutaan myös cmd-tiedostoa (build_23_path), jonka polku haetaan YAML-tiedostosta. Cmd-tiedostossa on CODESYS-ohjelmalle annettavat käännöskomennot, jotka on esitelty aiemmin kuvassa 6.

```
else:
    build_23_module = f'"{codesys23_path}" {project_path} /show hide' \
                      f' /cmd "{build_23_path}"' \
                      f' /out {module_name}.log'
    args = shlex.split(build_23_module)
    proc = subprocess.Popen(args)
```

Kuva 12. CODESYS 2.3 -käännöskomento.

Julkaisupakettiin sisältyy useita tiedostoja ja kansioita versionhallinnasta sekä binääritiedostoja, jotka generoituvat moduulien käännöksistä. Kaikki tiedostot, polut sekä julkaisupaketin nimi ja versionumero on määritelty YAML-tiedostoon, josta ne haetaan Pythoniin. Tiedostot kopioidaan Pythonilla shutil-kirjastoa käyttäen. Kuvassa 13 havainnollistetaan, kuinka YAML-tiedosto käydään läpi. Kopioinnit suoritetaan tiedostoon määriteltyjen polkujen mukaisesti.

```
# Copy files to package
with open(build_config) as f:
    data = yaml.load(f, Loader=yaml.FullLoader)
    dependencies = data['Dependencies']
    package_name = data['package_name']
    release_package_name = data['release_package_name']
    version = data['version']
    zip_password = data['zip_password']
    zip_destination = data['zip_destination']
    work_folder = data['work_folder']
    source_code_folder = data['source_code_folder']

for files in dependencies:
    src = files['source_location']
    dst = files['destination_location']
    if os.path.isdir(src):
        if 'exclude' in files:
            exclude = files['exclude']
            copytree(src, dst, ignore=ignore_patterns(*exclude), dirs_exist_ok=True)
        else:
            copytree(src, dst, dirs_exist_ok=True)
    elif os.path.isfile(src):
        copy(src, dst, follow_symlinks=True)
    else:
        print('{} file/folder does not exist'.format(dependencies['source_location']))
```

Kuva 13. YAML-tiedoston läpikäynti ja tiedostojen kopiointi.

Kun kaikki julkaisupaketin sisältämät tiedostot ja kansiot on kopioitu, ne pakataan zip-muotoon ja zip-paketille asetetaan nimi ja salasana. Zip-paketin lisäksi julkaisupakettiin kopioidaan projektin lähdekoodit (workdir-kansio). Lopuksi lähdekoodit ja pakattu zip-tiedosto pakataan vielä kerran. Pakkaukset tehdään komentoriviltä subprocess-kirjastoa apuna käyttäen. Kuvassa 14 näytetään, kuinka paketointi on tehty.

```
# Zip package files
zip_command_password = '"C:\\Program Files\\7-Zip\\7z.exe" a -tzip -r -p'
zip_command = '"C:\\Program Files\\7-Zip\\7z.exe" a -tzip -r'

zip_files = f'{zip_command_password}{zip_password}' \
            f' "{zip_destination}\\{package_name}{version}.zip"' \
            f' "{work_folder}\\*.*"'
subprocess.run(zip_files, shell=True)

# Zip release package
zip_release_package = f'{zip_command}' \
                      f' "{zip_destination}\\{release_package_name}{version}.zip"' \
                      f' "{source_code_folder}\\*.*" "{zip_destination}"\\{package_name}{version}.zip"'
subprocess.run(zip_release_package, shell=True)
```

Kuva 14. Julkaisupaketin paketointi zip-tiedostoksi.

Projektissa on kaksi Excel-tiedostoa, joista generoidaan julkaisupakettiin konetyyppiparametrit ja kielitiedostot. Generointi tapahtuu Excel-tiedostoista nappia painamalla. Projektitiimin toiveena oli, että parametrien ja kielitiedostojen generointi myös automatisoitaisiin. Automatisointi tapahtui niin, että tehtiin Excel-makro, joka painaa haluttua nappia. Pythonilla on mahdollista ajaa Excelissä olevia makroja win32-kirjaston avulla. Kuvassa 15 on esitetty, kuinka makro ajettiin Pythonin avulla. YAML-tiedostosta haettiin polku Excel-tiedostoon sekä makron nimi, joka painaa generointinappia.

```
with open(build_config) as f:
    data = yaml.load(f, Loader=yaml.FullLoader)
    machine_type_params_path = data['machine_type_params_excel_path']
    machine_type_params_macro = data['machine_type_params_macro']

excel = win32.Dispatch("Excel.Application")  # create an instance of Excel
book = excel.Workbooks.Open(Filename=f'{machine_type_params_path}')
excel.Run(f'{machine_type_params_macro}')  # This runs the macro
book.Save()
book.Close()
excel.Quit()
```

Kuva 15. Tiedoston generointi Excelistä.

Jokainen julkaisupaketin kokoamisen vaihe on jaettu metodeihin. Käyttäjä voi hallinnoida metodeja komentoriviparametreja käyttäen. Python-koodiin lisättiin komentorivikomennon tarkastelu, jonka perusteella suoritetaan halutut toiminnot. Kuvassa 16 on esitelty komentoriviparametrien käyttö. Komentorivikomennossa kerrotaan suoritettava Python-skripti, ”-c”- tai ”—config” -parametrin jälkeen kerrotaan YAML-konfiguraatiotiedoston polku, sekä suoritettavia metodeja vastaavat parametrit.

```
"""This module contains tool to build and package Codesys release.

-c parameter is used to set path to build config yaml
-b parameter runs build for package specified in config.yaml.
-r parameter collects release deliverables and creates release zip file.
-p parameter generates machine type parameters.
-l parameter generates language files.

Typical usage example:

    python release_tool.py -c config.yaml -b -r
    python release_tool.py --config config.yaml -b
"""
```

Kuva 16. Komentoriviparametrien käyttö.

Kuvasta 17 selviää, kuinka tarkastelu tehtiin. Jos komentorivikomennossa on käännöskomentoa vastaava parametri ja YAML-tiedoston polku on määritelty, suoritetaan ”build_modules” -metodi.

```
    # Check if build parameter is set
    for index, arg in enumerate(sys.argv):
        if arg in ['--build', '-b'] and yaml_file is not None:
            build_modules(yaml_file)
            del sys.argv[index]
            break
```

Kuva 17. Komentoriviparametrien tarkastelu.

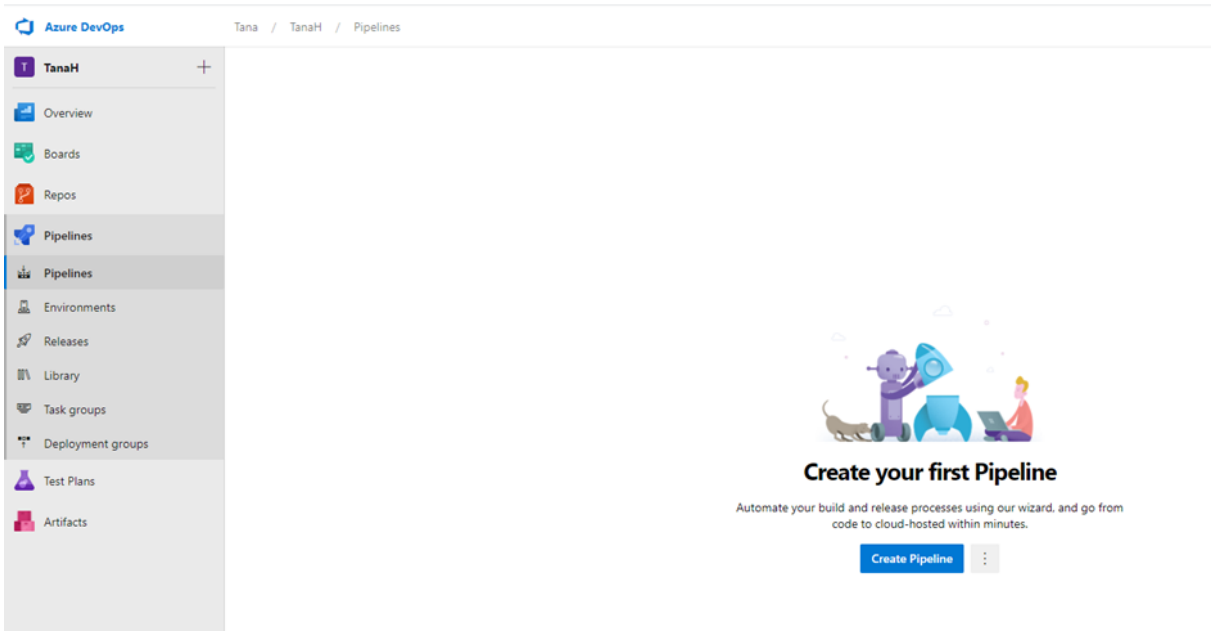
4.5 Agentin käyttöönotto

Agenttina päätettiin käyttää itse isännöityä agenttikonetta. Agenttikone on Epecin palvelimella toimiva virtuaalikone. Agenttikonetta voi hallinnoida etätyöpöytäyhteydellä. Agenttikoneeseen asennettiin tarvittavat ohjelmat ja työkalut. CODESYS-versiot 2.3 ja 3.5, paketointiohjelma 7zip, Python-versio 3.8 ja Office 365. Kaikki tehdyt työkalut tuotiin agenttikoneelle ja niitä käytetään Pipelinen kautta. Eli kaikki työ, mitä tarvitaan julkaisupaketin tekemiseen, on ulkoistettu agentille. Pipelinen käyttö ei siis kuormita käyttäjän konetta lainkaan.

4.6 Pipelinen pystytys Azure DevOpsiin

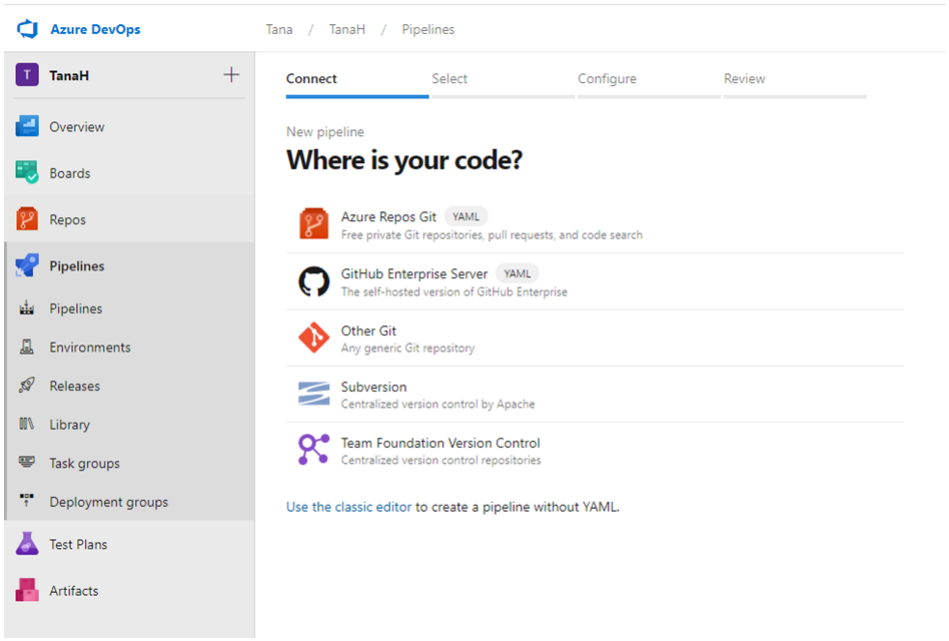
Azure DevOps -palvelu on kaikille ilmainen, mutta sinne kirjautuminen vaatii Microsoft-tilin. Microsoft-tili on mahdollista luoda kirjautumisen aikana. Kirjautumisen jälkeen käyttäjälle luodaan organisaatio, jos tiliä ei ole jo liitetty organisaatioon.

Organisaatiosta löytyvät kaikki Azure DevOpsissa olevat projektit, jotka sisältävät kaikki tarvittavat toiminnallisuudet. Toiminnallisuuksien käyttö määritellään projektikohtaisesti. Seuraavaksi valitaan projekti, jolle Pipeline halutaan ottaa käyttöön. Tässä tapauksessa Pipeline toteutettiin Tana H-projektille. Uuden Pipelinen käyttöönotto tapahtui kuvan 18 mukaisesti.



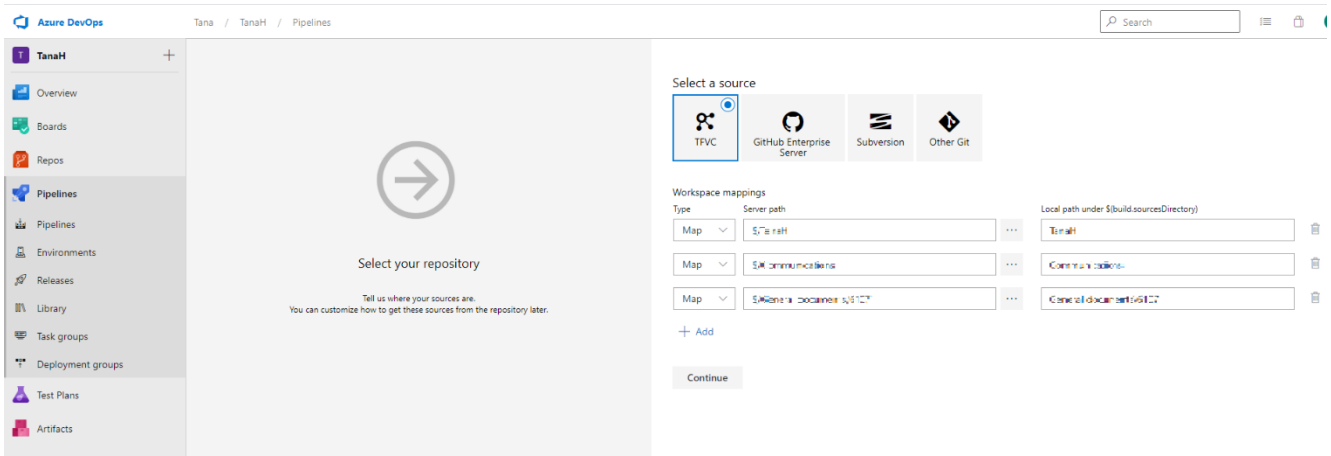
Kuva 18. Uuden Pipelinen luonti.

Pipelinen määrittelyn ensimmäinen vaihe oli valita versionhallintajärjestelmä (Kuva 19), jossa projektin koodit ja tiedostot sijaitsevat. Tässä projektissa käytettiin Team Foundation Version Controlia.



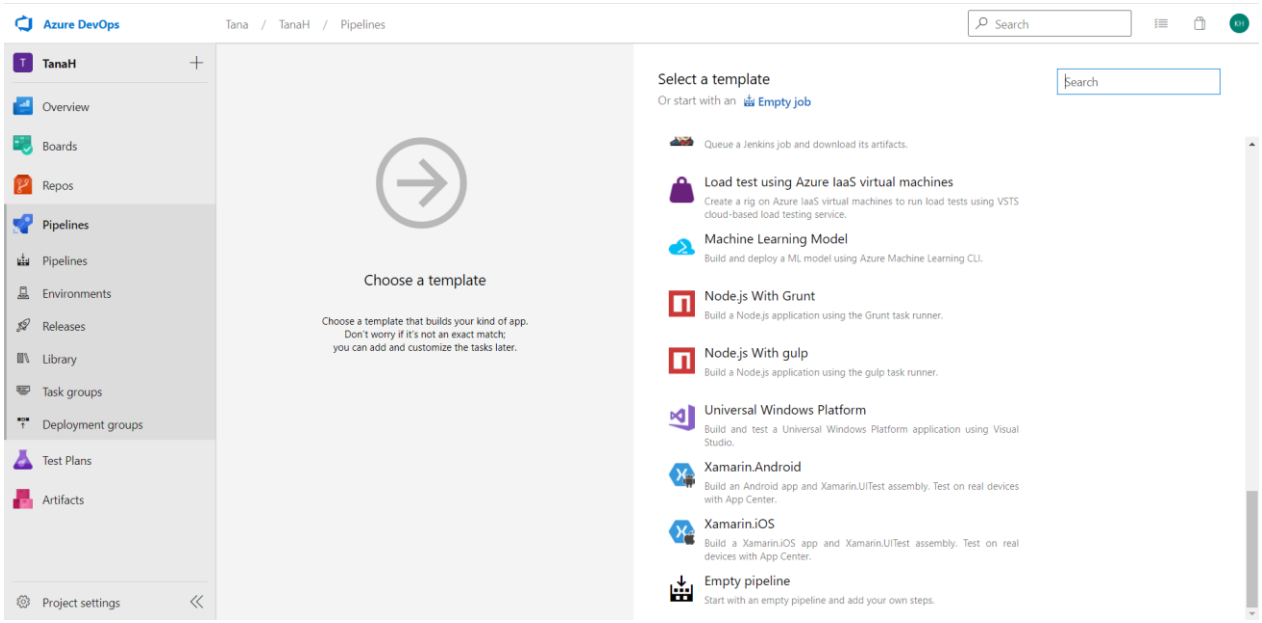
Kuva 19. Versionhallintajärjestelmän valinta.

Seuraavaksi valittiin kaikki versionhallinnan haarat, joissa sijaitsee Pipelinen käytön kannalta oleellisia tiedostoja. Tämän opinnäytetyön tapauksessa oleellisia olivat kaikki haarat, joissa oli julkaisupakettiin tarvittavia tiedostoja. Haarat lisättiin kuvan 20 mukaisesti. Kaikkiin Pipelinen asetuksiin voi palata ja niitä voi muuttaa myöhemmin.



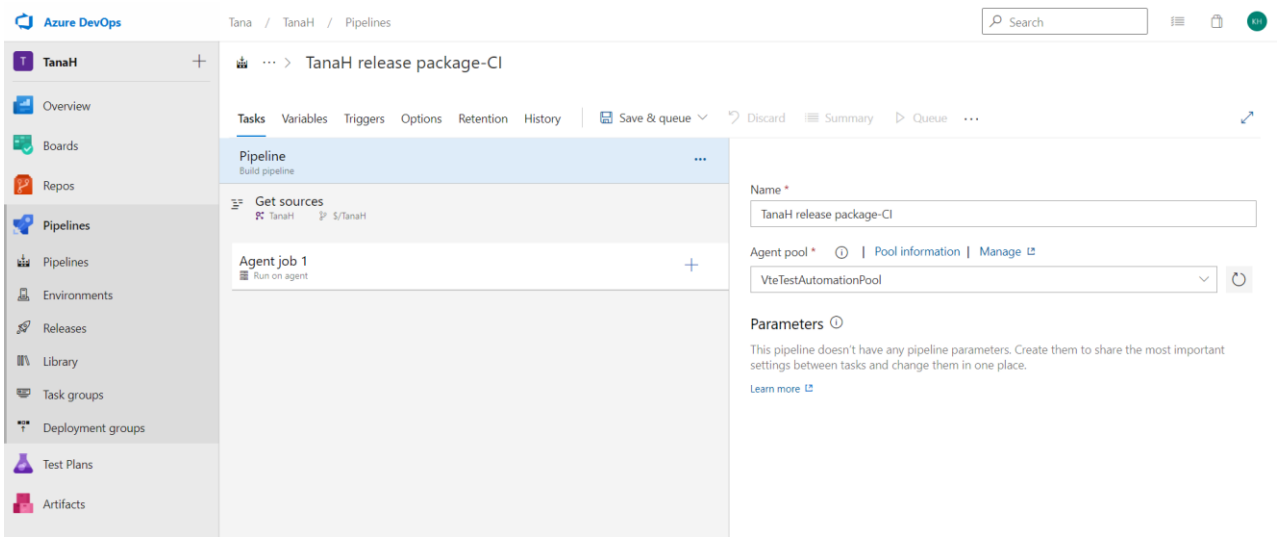
Kuva 20. Versionhallinnan haarojen valinta.

Versionhallinnan asetusten jälkeen on mahdollista valita valmis mallipohja (Kuva 21). Mallipohja luo automaattisesti tarvittavat työvaiheet eri projektityypeille, joita voi helposti muokata omaan projektiin sopivaksi. CODESYS-projektille tällaista valmista mallipohjaa ei ollut, joten valittiin tyhjä pohja (Empty pipeline).



Kuva 21. Pipelinen pohjan valinta.

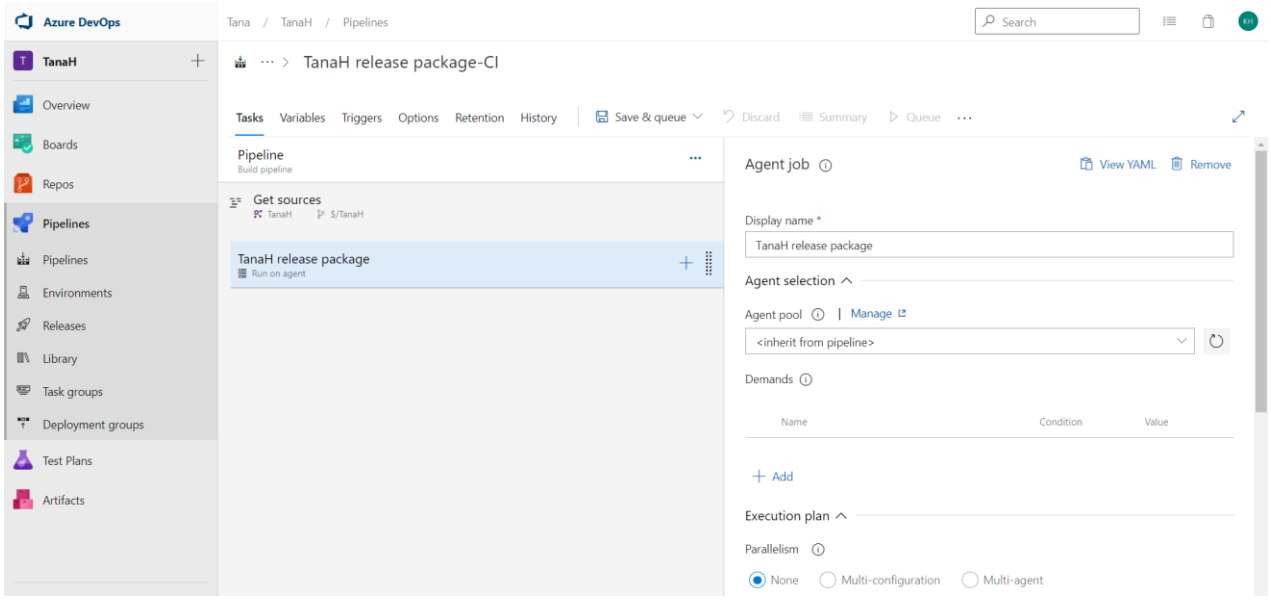
Seuraavassa vaiheessa nimettiin Pipeline ja valittiin käytettävä agenttiympäristö. Tässä kohtaa olisi mahdollista käyttää myös Microsoftin isännöimää agenttiympäristöä. Tämä projekti käytti kuitenkin itse isännöityä agenttiympäristöä, joka oli nimeltään ”VteTestAutomationPool” (Kuva 22).



Kuva 22. Agenttiympäristön valinta.

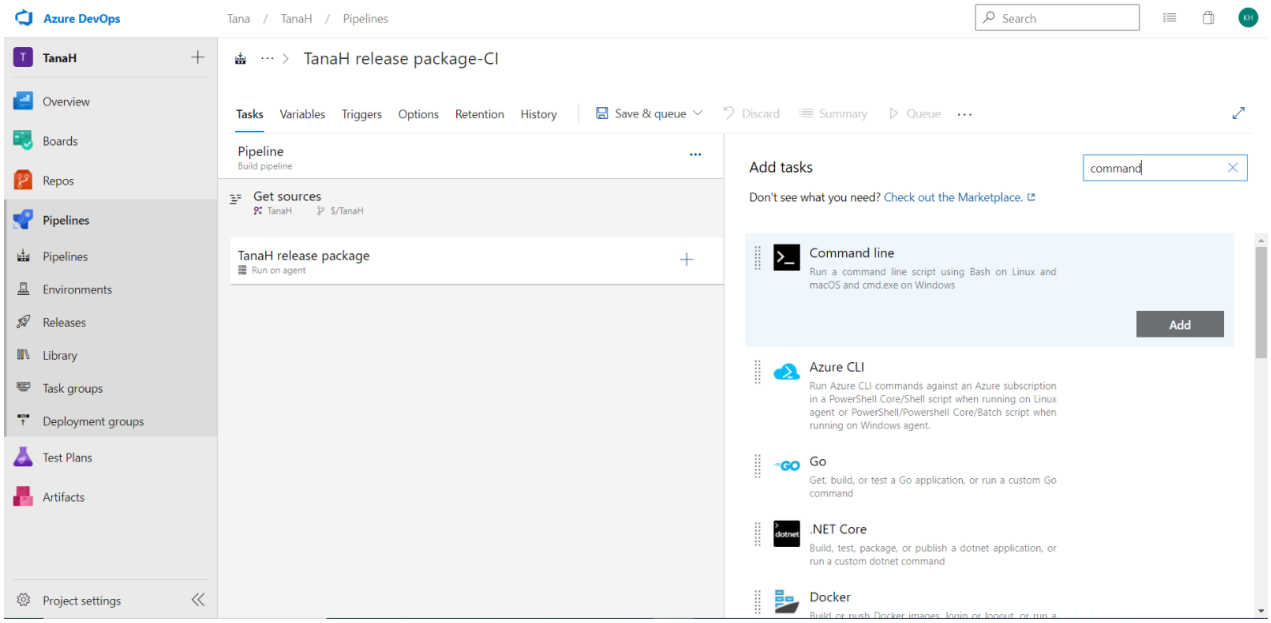
Kuvassa 23 näkyy vaadittavan agentin määrittely. Jos käytettäviä agentteja olisi useita, tähän Pipelineen vaadittavan agentin ominaisuudet voisi listata tässä ikkunassa. Demands-kohdan

alle luetellaan tarvittavat ominaisuudet. Pipeline osaa etsiä agenttiympäristöstä työhön kelpaavan agentin. Jos agentti on varattuna toisessa tehtävässä, työ asettuu jonoon.



Kuva 23. Agentin määrittely.

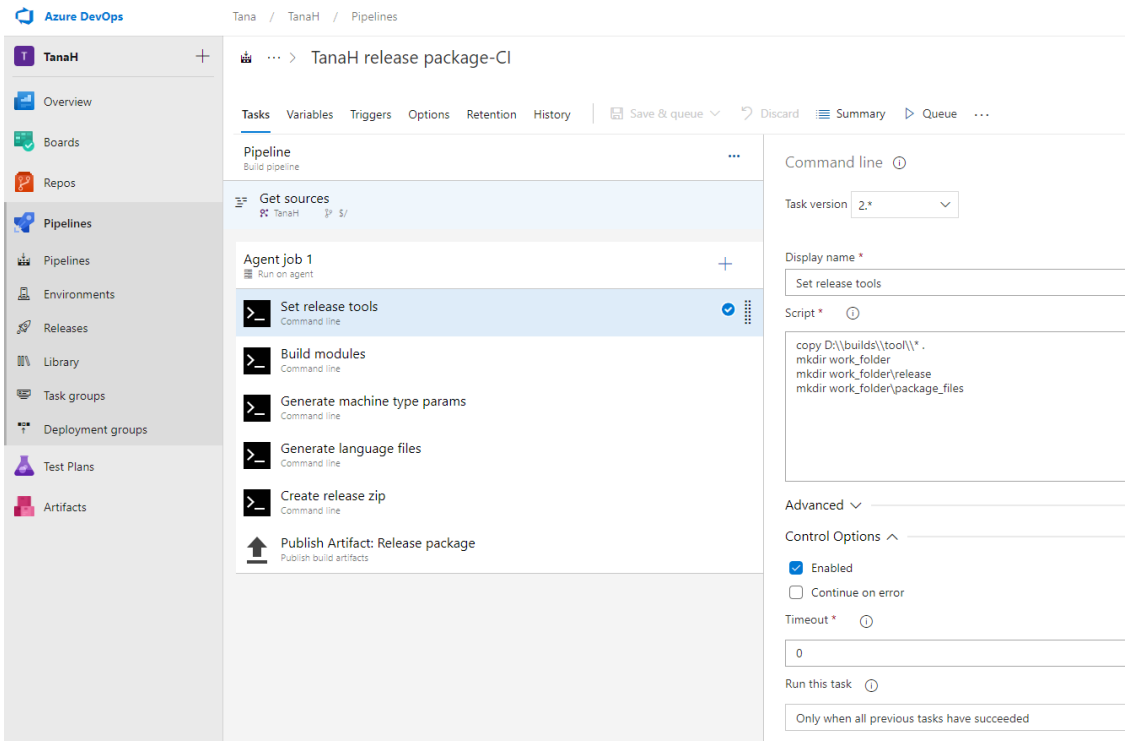
Kaikki Pipelinen ja agentin asetukset on nyt määritelty. Seuraava vaihe oli lisätä agentin tehtävät. Kuvassa 24 näytetään, kuinka tehtävät lisätään. Julkaisupaketin kokoamiseen tehdyn työkalun käyttö tapahtuu komentoriviltä, joten käytettiin ”Command line” -toimintoja.



Kuva 24. Agentin tehtävien valinta.

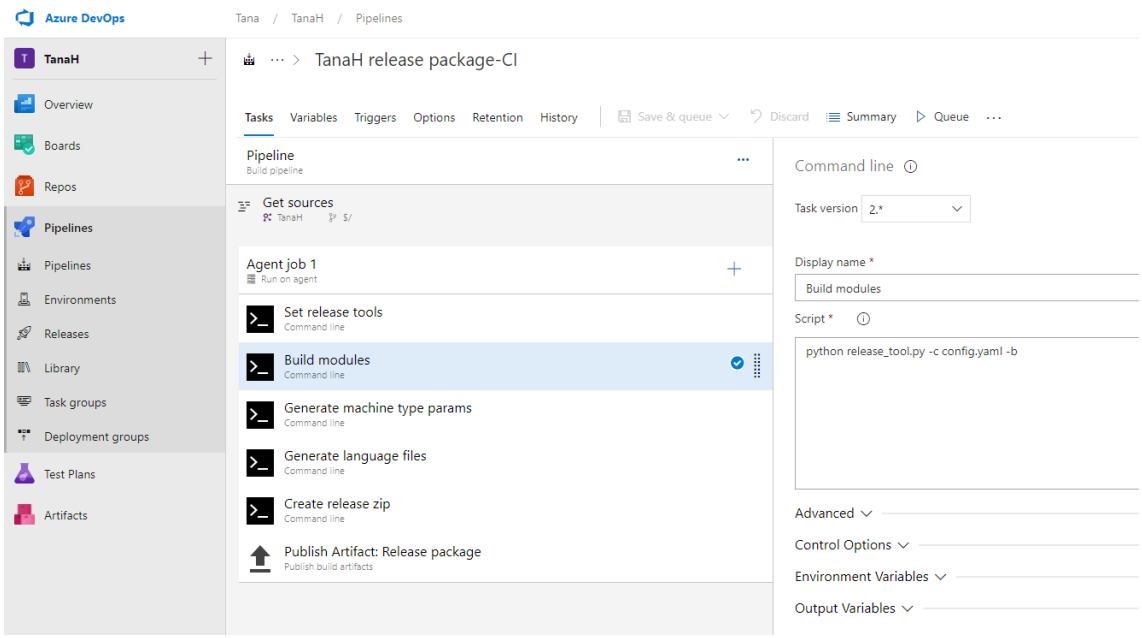
Ensimmäinen työvaihe oli alustaa tarvittavat työkalut (Kuva 25). Kaikki työkalut ja YAML-tiedosto kopioitiin agenttikoneella oikeaan paikkaan, josta ne voidaan suorittaa. Työkalujen kopioinnin jälkeen tehtiin kansio, jonka sisään tehtiin ”release”- ja ”package_files”-kansiot. Nämä kansiot ja polut on määritelty myös YAML-konfiguraatiotiedostoon. Python-skripti kopioi julkaisupaketin tiedostot package_files -kansioon, jonka jälkeen se muodostaa zip-pakatun paketin ”release”-kansioon. Myöhemmin paketti kopioidaan tuosta kansiosta Pipelinelle.

Jokaisen tehtävän ohjausasetuksiin (Control options) laitettiin: ”Only when all previous tasks have succeeded” -vaihtoehto. Eli tehtävä suoritetaan vain, jos kaikki edelliset tehtävät on suoritettu onnistuneesti.



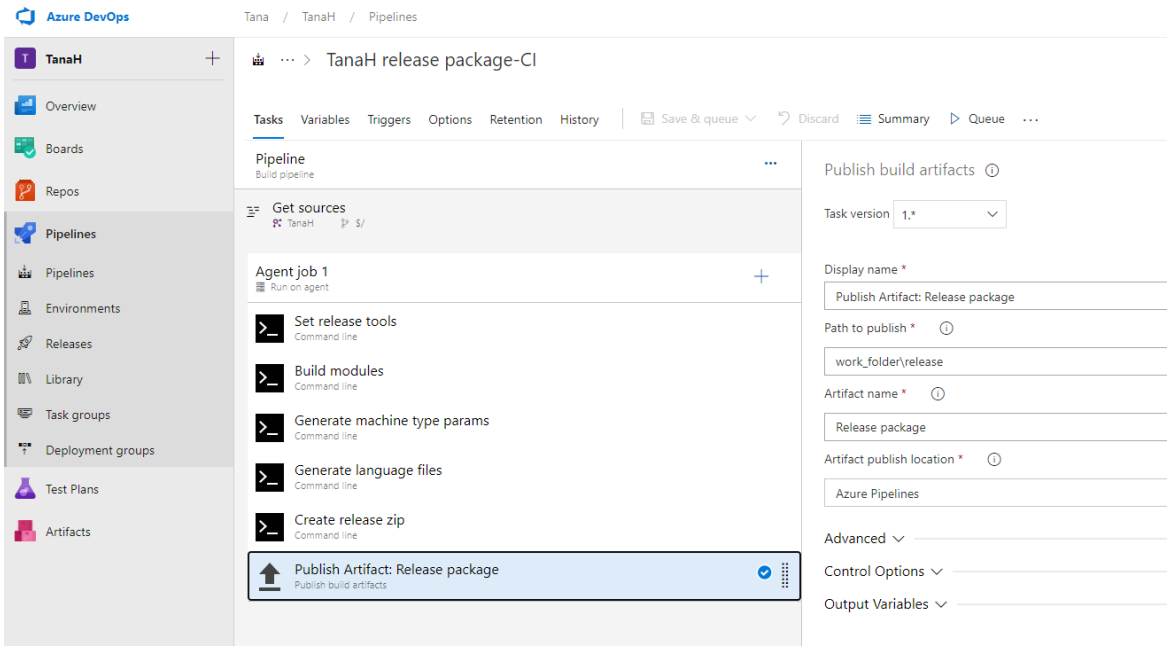
Kuva 25. Työkalujen alustus.

Seuraavat tehtävät olivat moduulien käännökset, konetyyppiparametrien ja kielitiedostojen generointi sekä julkaisupaketin kokoaminen zip-muotoon. Nämä kaikki tehtävät suoritetaan komentorivikomennoilla, joissa vaihtuvat vain komentoriviparametrit (Kuva 26). Esimerkiksi moduulien käännöksiin käytetty komentoriviparametri on -b, konetyyppiparametrien generointiin -p jne. Kaikki komentoriviparametrit on käsitelty aiemmin tässä luvussa.



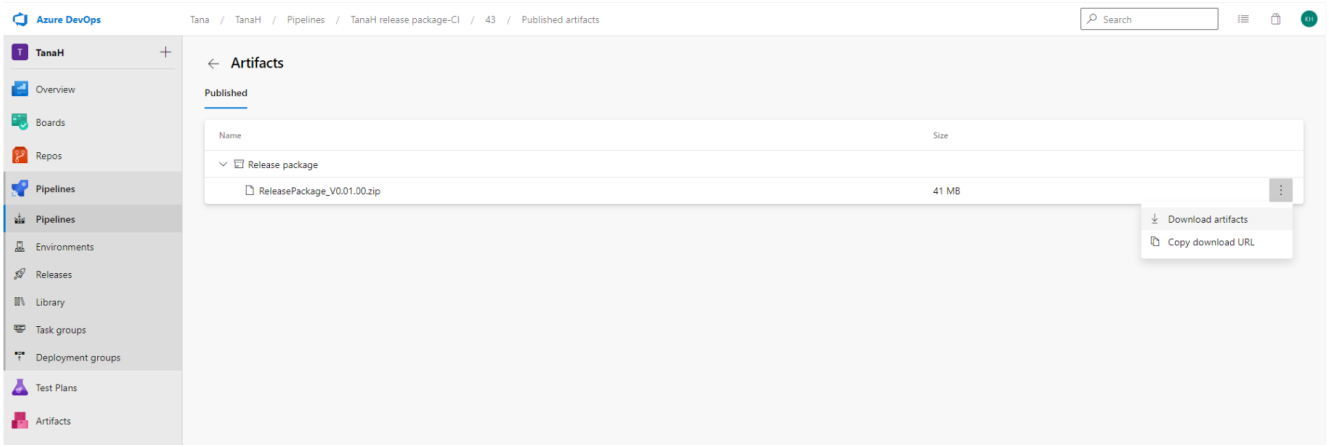
Kuva 26. Käännöskomento.

Paketin julkaisussa Pipelinelle käytettiin Publish Artifact -toimintoa (Kuva 27). Toiminnolle määriteltiin julkaistavan paketin polku, artefaktin nimi ja julkaisupaikka.



Kuva 27. Paketin julkaisu Pipelinelle.

Tässä vaiheessa kaikki työvaiheet oli määritelty ja Pipeline on käyttövalmis. Onnistuneen Pipelinen ajon jälkeen Pipelinelle generoitui julkaisupaketti, jonka sisältö vastasi YAML-konfiguraatiotiedoston sisältöä (Kuva 28). Julkaisupaketin voi ladata omalle koneelle, tai siitä voi kopioida latauslinkin.



Kuva 28. Julkaisupaketti.

5 YHTEENVETO JA JATKOKEHITYS

Tämän opinnäytetyön tavoitteena oli analysoida modernin Pipeline-toteutuksen mahdollisuudet ja toteuttaa asiakasprojekteja varten Pipeline, CODESYS-pohjaisten ohjelmistojulkaisujen kokoamista varten. Tarkoituksena oli myös toteuttaa Pipelinen työkalut niin, että ne on helppo ottaa käyttöön myös muissa asiakasprojekteissa.

Työ päätettiin toteuttaa Tana H-sarjan kaatopaikkajyrän ohjelmistoprojektiin. Tähän projektiin päädyttiin, koska asiakkaalla oli kiinnostusta ottaa käyttöön automaattinen julkaisupaketin kokoaminen ja myöhemmin myös muita CI/CD Pipelinen toimintoja. Toteutus alkoi tutustumalla julkaisupaketin manuaaliseen kokoamiseen. Ensimmäinen automatisoitava vaihe oli ohjausyksiköiden koodien kääntäminen. Automatisointi onnistui komentorivikomennoilla. Ongelmia tässä tuotti kahden eri CODESYS-version käyttö. Komentorivikomennot olivat erilaisia eri CODESYS-versioille. Käännösten jälkeen seuraava vaihe oli julkaisupaketin kokoamisen automatisoiminen. Tällöin päätettiin ohjelmoida Pythonilla työkalu, joka kopioi, generoi ja paketoi julkaisupaketin tiedostot zip-muotoon. Tämä työkalu käyttää apunaan YAML-konfiguraatiotiedostoa, johon määritellään kaikki julkaisupakettiin tulevat tiedostot ja niiden sijainnit. Koska kaikki julkaisupaketin kokoamisen kannalta oleelliset asiat on tuotu konfiguraatiotiedostoon, voidaan Python-työkalua käyttää muissakin projekteissa. Riittää, että muokataan konfiguraatiotiedosto projektille sopivaksi.

Kaikkien työkalujen ollessa valmiita oli seuraava vaihe pystyttää Pipeline Azure DevOpsiin. Agenttina, eli Pipelinen ”työntekijänä”, päätettiin käyttää itse isännöityä agenttia, koska se antoi paremmat mahdollisuudet ohjelmien asennuksiin ja niiden hallintaan. Agenttikoneelle asennettiin kaikki tarvittavat ohjelmat, sekä sille tuotiin tehdyt työkalut. Seuraavaksi määriteltiin Pipeline, joka tekee agentin avustuksella Tana H-projektin julkaisupaketin.

Lopputulokseksi saatiin Pipeline, joka kääntää ohjelmistokoodin ja kokoaa julkaisupaketin yhtä nappia painamalla. Pipelineen tehdyistä työkaluista saatiin tehtyä niin yleiskäyttöiset kuin mahdollista, jotta työkaluja voidaan käyttää useissa projekteissa. Joitakin pieniä muutoksia Python-työkaluun voi joutua tekemään, jos esimerkiksi tiedostojen paketointi tehdään eri tavalla.

Tulevaisuudessa Pipelinea tullaan varmasti jatkokehittämään. Julkaisupaketti tullaan tallentamaan Azure Artifactsiin, josta se voidaan jakaa esimerkiksi testausympäristöön.

Paketin salasana määritellään tällä hetkellä YAML-tiedostossa, mutta tulevaisuudessa sen voisi piilottaa esimerkiksi Azure Key Vaultin avulla. Työkalut tullaan siirtämään pois agenttikoneelta versionhallintaan, josta agentti käy ne hakemassa. Myös YAML-tiedosto tullaan siirtämään versionhallinnan projektitasolle. Työkalujen sijainnit on määritelty YAML-tiedostoon, mutta ne tullaan tulevaisuudessa siirtämään agentin asetuksiin, pois YAML-tiedostosta. Näin YAML-tiedostosta saataisiin mahdollisimman selkeä ja projektikohtainen. Lisäksi jatkuva julkaisu/toimitus (continuous delivery/deployment) on asia, jota tullaan ottamaan käyttöön, esimerkiksi julkaisupaketin automaattinen lataus testiympäristöön. Myös automaattiset testit, yksikkötestit ja koodiskannerin käyttöönotto ovat potentiaalisia jatkokehityskohteita.

LÄHTEET

Brown D. 2015. What is DevOps? [Verkkojulkaisu]. [Viitattu 10.2.2021]. Saatavilla: https://www.donovanbrown.com/post/what-is-devops

CODESYS. Ei päiväystä. Product news. [Verkkosivu]. CODESYS GmbH. [Viitattu 13.2.2021]. Saatavana: https://www.codesys.com/

CODESYS. Ei päiväystä. The Company. [Verkkosivu]. CODESYS GmbH. [Viitattu 14.2.2021]. Saatavana: https://www.codesys.com/company.html

Devopedia. 2020. Continuous Integration. [Verkkojulkaisu]. Devopedia. [Viitattu 24.2.2021]. Saatavilla: https://devopedia.org/continuous-integration#Wikipedia-2018

Epec. Ei päiväystä. Company. [Verkkojulkaisu]. Epec Oy. [Viitattu 6.2.2021]. Saatavilla: https://epec.fi/company/

Epec. Ei päiväystä. EPEC 3724 CONTROL UNIT. [Verkkosivu]. Epec Oy. [Viitattu 20.2.2021]. Saatavana: https://epec.fi/products/control-system-products-3724/

Epec. Ei päiväystä. EPEC 5050 CONTROL UNIT. [Verkkosivu]. Epec Oy. [Viitattu 21.2.2021]. Saatavana: https://epec.fi/products/control-system-products-5050/

Epec. Ei päiväystä. EPEC 6107 DISPLAY UNIT. [Verkkosivu]. Epec Oy. [Viitattu 22.2.2021]. Saatavana: https://epec.fi/products/display-6107/

Epec. Ei päiväystä. Epec company presentation. [Verkkojulkaisu]. Epec Oy. [Viitattu 5.2.2021]. Saatavilla: Vain yrityksen sisäisessä käytössä.

Fowler M. 2006. Practices Of Continuous Integration. [Verkkojulkaisu]. [Viitattu 24.2.2021]. Saatavilla: https://www.martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration

Fowler M. 2013. ContinuousDelivery. [Verkkojulkaisu]. [Viitattu 30.12.2020]. Saatavana: https://martinfowler.com/bliki/ContinuousDelivery.html

Guckenheimer S. 2017. What is Continuos Integration? [Verkkojulkaisu]. [Viitattu 30.12.2020]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration

Innovative Defense Technologies. Ei päiväystä. What is Automated Software Testing? [Verkkojulkaisu]. [Viitattu 24.2.2021.] Saatavilla: https://idtus.com/what-we-do/what-is-automated-software-testing/

Korpela, J. 2001. Perustietoa Python-ohjelmointikielestä. [Verkkojulkaisu]. [Viitattu 15.2.2021]. Saatavana: http://jkorpela.fi/python/

Microsoft Azure. 2018. Deep dive into Azure Boards. [Verkkojulkaisu]. Microsoft Corp. [Viitattu 10.2.2021]. Saatavana: https://azure.microsoft.com/en-in/blog/deep-dive-into-azure-boards/

Microsoft. 2019a. Whay is Azure Pipelines? [Verkkojulkaisu]. Microsoft Corp. [Viitattu 11.2.2021]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

Microsoft. 2019b. Exploratory and manual testing scenarios and capabilities. [Verkkojulkaisu]. Microsoft Corp. [Viitattu 11.2.2021]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/test/overview?view=azure-devops

Microsoft. 2020a. Artifacts in Azure Pipelines. [Verkkojulkaisu]. Microsoft Corp. [Viitattu 10.2.2021]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/pipelines/artifacts/artifacts-overview?view=azure-devops

Microsoft. 2020b. What is Azure Boards? [Verkkojulkaisu]. Microsoft Corp. [Viitattu 10.2.2021]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/boards/get-started/what-is-azure-boards?view=azure-devops&tabs=agile-process

Microsoft. 2020c. What is Azure Repos? [Verkkojulkaisu]. Microsoft Corp. [Viitattu 11.2.2021]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/repos/get-started/what-is-repos?view=azure-devops

Microsoft. 2020d. About agents & agent pools. [Verkkojulkaisu]. Microsoft Corp. [Viitattu 24.2.2021]. Saatavana: https://docs.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops&viewFallbackFrom=azdevops&tabs=browser

Microsoft. 2021. What is DevOps? [Verkkosivu]. Microsoft Corp. [Viitattu 11.2.2021]. Saatavilla: https://azure.microsoft.com/en-us/overview/what-is-devops/

Microsoft. Ei päiväystä. Azure Artifacts. [Verkkosivu]. Microsoft Corp. [Viitattu 10.2.2021]. Saatavilla: https://azure.microsoft.com/en-us/services/devops/artifacts/

Muffatto, M. 2006. Open Source: A Multidisciplinary Approach. [Verkkokirja]. London: Imperial Collage Press. [Viitattu 10.3.2021]. Saatavana ProQuest Ebook Central -palvelusta. Vaatii käyttöoikeuden.

Outlaw R. 2017. What is version control? [Verkkojulkaisu]. [Viitattu 24.2.2021]. Saatavilla: https://docs.microsoft.com/en-us/azure/devops/learn/git/what-is-version-control

Python. Ei päiväystä. About. [Verkkosivu]. Python Software Foundation. [Viitattu 15.2.2021]. Saatavana: https://www.python.org/about/

Python. Ei päiväystä. Python Software Foundation. [Verkkosivu]. Python Software Foundation. [Viitattu 16.2.2021]. Saatavana: https://www.python.org/psf/

Robson, S. 2013. Agile SAP: Introducing Flexibility, Transparency and Speed to SAP Implementations. [Verkkokirja]. IT Covermance Ltd. [Viitattu 19.3.2021]. Saatavana ProQuest Ebook Central -palvelusta. Vaatii käyttöoikeuden.

Rossel, S. 2017. Continuous Integration, Delivery and Deployment. [Verkkokirja]. Birmingham: Packt Publishing Ltd. [Viitattu 1.3.2021]. Saatavana: ProQuest Ebook Central -palvelusta. Vaatii käyttöoikeuden.

Tana. Ei päiväystä. Ainutlaatuinen TANA-konstruktio. [Verkkosivu]. Tana Oy. [Viitattu 20.2.2021]. Saatavana: https://tana.fi/fi/tana-tuotteet/tana-kaatopaikkajyra/h-sarja/

Yaml. Ei päiväystä. YAML Ain’t Markup Language. [Verkkosivu]. [Viitattu 2.3.2021]. Saatavilla: https://yaml.org/