jamk.fi

Versionhallinnan käyttöönotto ohjelmistoalan yrityksessä

Jaska Ahlfors

Opinnäytetyö
Maaliskuu 2021
Tietojenkäsittely ja tietoliikenne
Insinööri (AMK), tieto- ja viestintätekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Ahlfors, Jaska	Opinnäytetyö, AMK	Maaliskuu 2021
	Sivumäärä 46	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Versionhallinnan käyttöönotto ohjelmistoalan yrityksessä

Tutkinto-ohjelma
Insinööri (AMK), tieto- ja viestintätekniikka

Työn ohjaaja(t)
Esa Salmikangas

Toimeksiantaja(t)

Tiivistelmä

Opinnäytetyön aiheena oli versionhallinnan käyttöönotto ohjelmistoalan yrityksessä. Tavoitteena oli selvittää tärkeimpiä kriteereitä versionhallintajärjestelmän valitsemiselle, sekä mitä muita asioita on hyvä selvittää ja suunnitella ennen kuin itse käyttöönotto voidaan tehdä. Tavoitteena oli myös luoda sisäinen dokumentaatio versionhallinnan käytölle ja perehdyttää henkilöstö sen käyttöön. Käyttöönotto oli tarkoitus suunnitella ja toteuttaa kuormittamatta yrityksen prosesseja käyttöönotto- tai perehdytysvaiheessa.

Tavoitteena oli myös tutkia, miten versionhallintaan voitaisiin integroida erilaisia sovelluksen elinkaarenhallintaan liittyviä työkaluja, kuten jatkuvan integraation ja jatkuvan julkaisun työkaluja.

Tutkimusmenetelmänä käytettiin työelämän kehittävää tutkimustoimintaa, tarkoituksena lähestyä yleistä ajankohtaista ongelmaa ja etsiä siihen yleishyödyllisiä ratkaisuja, hyödyntäen niitä toimeksiantajan prosessien kehittämisessä.

Tutkimuksen tuloksena saatiin toimeksiantajalle otettua käyttöön versionhallintajärjestelmä, sekä luotua sen käytölle organisaation sisäiset standardit ja strategiat. Versionhallintaan saatiin myös integroitua työkaluja sovelluksen elinkaaren hallintaan, muun muassa vaatimusten-, vian- ja testienhallintaan. Jatkuvasta integraatiosta ja jatkuvasta julkaisusta saatiin tuloksia jatkokehitystä ja myöhempää käyttöönottoa varten.

Toteutuksessa onnistuttiin hyvin. Oikea alusta ja palveluntarjoaja versionhallinnalle saatiin valittua. Käyttöönotto ja henkilöstön perehdytys toteutettiin ilman negatiivista vaikutusta prosesseihin. Sovelluksen elinkaaren hallinnan työkaluja saatiin otettua käyttöön. Versionhallinnan ja integroitujen työkalujen havaittiin tehostavan toimeksiantajan prosesseja.

Avainsanat (asiasanat)

Versionhallinta, ohjelmistokehitys, sovelluksen elinkaaren hallinta

Muut tiedot (Salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Ahlfors, Jaska	Bachelor's thesis	March 2021
		Language of publication: Finnish
	Number of pages: 46	Permission for web publication: x

Title of publication

Deployment of version control in a software company

Degree programme

Bachelor of Engineering, Information and Communication Technology

Supervisor(s)

Salmikangas, Esa

Assigned by

Abstract

The subject of the thesis was the deployment of version control in a sofware company. The goal was to find out the most important criteria for choosing the version control system and what other details there are to examine and plan before the deployment. One goal was also to create the documentation train the personnell to use the version control system. The deployment was to be carried out without hindering the existing processes.

One goal was to examine differerent tools related to application lifecycle management, including tools of continuous integration and continuous delivery.

Research-based developement assignment was used as the thesis implementation method. The aim was to approach a common problem and to find solutions that could be commonly utilized and would benefit the existing processes.

As the result of the research, a version control system was succesfully deployed. Functioning standards and strategies were also created for the internal use in organization. Tools for application lifecycle management were also integrated into the version control system. The tools included ways to control requirements, defects and tests. Research of continuous integration and continuous delivery resulted information that would benefit further research and later use.

Execution phase was succesful. Right platfrom and provider was chosen for the version control. The deployment and the training were conducted without negative impact on the existing processes. Tools for application lifecycle management were utilized. The deployment of the version control system and the tools integrated in it had an positive impact on the processes.

Keywords/tags (subjects)

Version control, software development, application lifecycle management

Miscellaneous (Confidential information)

Sisältö

Kuviot

1 Tutkimuksen lähtökohdat

Aiheesta yleisesti

Versionhallinta hallitsee muutoksia tiedostoihin ja ohjelmistoihin. Versionhallinnalla voidaan muun muassa tarkastella tiedostojen muutoshistoriaa, palauttaa niitä aiempaan tilaansa ja yhdistellä tiedostoihin eri tahoilla tehtyjä muutoksia. Ohjelmistokehityksessä versionhallintaa käytetään ohjelmistojen eri versioiden ylläpitämiseen, sekä useamman kehittäjän samanaikaisen työskentelyn mahdollistamiseen saman projektin parissa.

Toimeksiantajan esittely

Toimeksiantaja on yksityinen automaatio- ja ohjelmistoalan yritys Kotkasta, joka on perustettu 2002. Ohjelmistoprojektien parissa ohjelmistokehittäjien määrä parissa on ollut pitkään yksi tai kaksi kehittäjää, mutta viimeaikoina henkilöstömäärä on ollut kasvussa.

Tutkimuksen tarkoitus

Tämän tutkimuksen tarkoitus kartoittaa ohjelmistoalalla työskentelevälle yritykselle optimaalisin versionhallinta-alusta, sekä suunnitella sen käyttöönotosta olemassaoleville projekteille mahdollisimman sujuva. Tarkoituksena on ensin kartoittaa, mitkä kriteerit ovat tärkeimpiä versionhallintajärjestelmän ja -alustan valitsemisessa, sekä mitä yrityksen projekteihin sekä prosesseihin liittyviä seikkoja tulee ottaa huomioon sen käyttöönottoa suunniteltaessa. Erilaisia artikkeileita, vertailuja ja ohjekirjoja versioonhallintaan liittyen on olemassa jo paljonkin, mutta laajempaa tutkimusta sen käyttöönottoon ja sitä edeltävään suunnitteluun ei ole löytynyt. Tämän tutkimuksen on tarkoitus tehdä asiasta mahdollisimman kokonaisvaltainen kartoitus, sekä samalla ottaa toimeksiantajalle käyttöön asianmukainen versionhallintajärjestelmä, tehostaen näin yrityksen prosesseja.

Kehittämistyön tarkoitus

Kehittämistyön tarkoitus on ottaa versionhallinta käyttöön toimeksiantajan projekteissa. Tavoitteena oli ennen kaikkea mahdollistaa useamman ohjelmistokehittäjän tehokas työskentely saman projektin parissa, mutta myös tarjota työkalut kattavamman ja automaattisemman versiohistorian ylläpitämiseen.

Samalla oli tarkoitus tutkia muita versionhallinnan mahdollistamia ja siihen integroitavia prosesseja parantavia työkaluja. Tarkasteltaviin kohtiin kuului erityisesti sovelluksen elinkaaren hallintaan liittyvien työkalujen hallintaan kuuluvien työkalujen yhdistäminen, sekä kuinka versionhallinnan kautta voitaisiin edistää ohjelmistotestausta ja kehittää testausautomaatiota versionhallinnan kautta.

Tärkeimpiä kysymyksiä, mihin työssä haettiin vastauksia, olivat versionhallintaratkaisujen tietoturva sekä prosessien tehokkuuden, sekä laadun turvaaminen. Tärkeitä avaintekijöitä olivat varmuuskopioiden taattu eheys, yrityksen tuotteen, eli ohjelmistokoodin suojaaminen ulkopuolisilta sekä ohjelmistoversioiden laadun varmistus sekä prosessien läpimenoaika.

Tutkimuksesta yleisesti

Opinnäytetyössä tutkittiin eri versionhallintajärjestelmiä ja niiden eroja keskenään. Tutkittavana ja vertailtavana oli myös eri versionhallinta-alustat, niiden palveluntarjoajat, sekä kysymys siitä, kannattaako versionhallintaa ylläpitää itse yrityksen omilla palvelimilla vai käyttää hyväkseen pilvipalveluita. Tutkimukseen sisältyi myös versionhallinnan parhaat käytänteet ja toimeksiantajan olemassa olevia käytäntöjen ja prosessien kartoitus ja sovittaminen yhteen niiden kanssa. Olennainen osa tutkimusta oli myös versionhallinnan käytön ohjeistuksen laatiminen ja perehdyttäminen sitä käyttävälle henkilökunnalle.

Toteutustapa

Tutkimusmenetelmänä käytettiin työelämän kehittävää tutkimustoimintaa. Tavoitteena oli tutkia ja kehittää toimeksiantajan prosesseja tavalla ja dokumentoida tutkimusta tavalla, josta on yleisesti hyötyä alalla.

2 Versionhallinta ja käsitteet

2.1 Versionhallinta yleisesti

Versionhallinnan avulla pidetään yllä tiedostojen muutoshistoriaa. Versionhallinnan tavoitteita on mahdollistaa tiedostojen palauttaminen aikaisempaan versioon ja useampien eri muutosten yhdistäminen yhteen samaan tiedostoon. (Somasundaram 2013.)

Versionhallintaan käytetään erilaisia tekniikoita ja erilaisia järjestelmiä. Yksinkertaisimmillaan versionhallinta voi olla tiedostokansioiden kopioimista ja uudelleennimeämistä versioiden mukaan. Kun projektit tai niiden kanssa työskentelevät ryhmät kasvavat suuremmiksi, tulee tästä tavasta hyvin työläs ylläpitää. Modernimmissa versionhallintatekniikoissa puhutaan muutosten tallettamista repositorioihin, eli tietovarastoihin. Ohjelmistotuotannossa käytetään pääasiassa käytännössä kolmea erilaista versionhallintajärjestelmää: paikallisia, keskitettyjä tai hajautettuja palvelimia. Paikallisessa palvelimessa tiedostot ovat kaikki työkoneella. Keskitetyissä palvelimissa repositorio, eli kaikki tiedostot ovat samalla palvelimella, johon kaikki muutokset tehdään. Hajautetuissa palvelimissa päärepositorio on yhdellä tai useammalla palvelimella ja jokaisella järjestelmän käyttäjällä on oma kopio siitä omalla koneellaan. (Somasundaram 2013.)

2.2 Versionhallinnan tarkoitus

Nykyään voisi melkein jopa sanoa, että ilman versionhallintaa suuremmat ohjelmistoprojektit ovat mahdottomia. Erityisesti on huomattu, että useamman kehittäjän työskenteleminen saman projektin parissa on ilman sitä vähintään työlästä, sillä muutosten yhdistelemiseen ja kirjaamiseen sekä useamman eri version ja niiden riippuvuuksien ylläpitäminen kuluu manuaalisesti hirveästi aikaa ja vaivaa. (Ahlfors 2020.)

Ohjelmistoalalla versionhallinta mahdollistaa vanhojen ohelmistoversioiden ylläpitämisen. Tästä on hyötyä esimerkiksi, jos eri ohjelmistojen kanssa tarvitsee käyttää eri versiota tietystä ohjelmasta, tai jos esimerkiksi epäonnistuneen ohjelmistopäivityksen jälkeen pitää palata vanhaan versioon tai etsiä ratkaisua toimimattomuuteen aiemmasta toteutuksesta. Samoin versionhallinta auttaa tuotantovaiheessa virhetilanteessa tiedostojen palauttaminen aiempaan hetkeen. Yksi toimivan versionhallinnan konkreettisista hyödyistä on myös mahdollisuus useammalle kehittäjälle työskennellä samojen tiedostojen parissa samanaikaisesti. Modernimmilla versionhallintamenetelmillä pysytään myös paremmin perillä muutoksista, sillä ne tallentavat parhaimmillaan tallentavat tietoa siitä mitä on muutettu, milloin ja kenen toimesta pelkän varmuuskopioinnin sijaan. (Somasundaram 2013.)

2.3 Käsitteet

2.3.1 SSH

SSH-protokolla on turvallinen ja yleisin metodi palvelimien etäkäyttöön. (Ellingwood 2014). SSH-protokollaa käytetään myös paljon versionhallinnassa ja koska sen turvallisuus on täten suuri osa versionhallinnan tietoturvaa, on tärkeää ymmärtää mihin se perustuu, jotta voidaan arvioida ja perustella versionhallintaratkaisun tietoturva organisaatiolle. SSH-protokollaan on hyvä perehtyä tarpeeksi ymmärtääkseen sen toimintamekanismin, mutta yksinkertaisuudessaan sen turvallisuus muodostuu useammasta tekijästä.

SSH-yhteys käyttää sekä symmetristä, että asymmetristä salausta. Symmetristä avainta käytetään salaamaan yhteys, kun taas asymmetristä salausta käytetään tunnistautumiseen. Palvelin ja yhteyttä muodostava laite muodostavat yhdessä symmetrisen avaimen, se on sessiokohtainen ja ulkopuolisille osapuolille tuntematon. (Ellingwood 2014.)

SSH-yhteyden eri vaiheissa käytetään asymmetristä salausta. Siinä käytetään julkista avainta (public key) ja yksityistä avainta (private key). Avaimilla on toisiinsa matemaattinen suhde, jonka ansiosta julkinen avain pystyy salaamaan viestejä, joiden salauksen ainoastaan sen parina toimiva yksityinen avain voi purkaa. Tämä toimenpide on yksisuuntainen, joten julkinen avain ei itsekään voi purkaa viestejä, joita on itse salannut. Yksityistä avainta ei myöskään voida mitenkään johtaa julkisesta avaimesta. Tämän takia, jos yhteyttä muodostava laite on onnistunut purkamaan salauksen, on sillä oltava hallussaan vaadittu avain.

Julkista avainta käytetään esimerkiksi, jos tunnistautuminen palvelimelle halutaan toteuttaa SSH-avaimella käyttäjätunnuksen ja salasanan sijaan. Palvelimelle tallennetaan kyseisen avainparin julkinen avain. Kun käyttäjä ottaa yhteyttä palvelimeen, palvelin lähettää julkisella avaimella salatun viestin. Käyttäjä todistaa henkilöllisyytensä pystymällä purkamaan avaimen. SSH-avaimen turvallisuutta voidaan koventaa suojaamalla yksityisen avaimen käyttö salasanalla. (Ellingwood 2014.)

2.3.2 Versionhallintahaarat

Versionhallintahaaran avulla kehittäjä voi erkaantua kehitettävästä versiosta luomalla työtilasta erillisen kopion, jolloin sen avulla voidaan kehittää jotain toista ominaisuutta, häiritsemättä muita projektin kanssa työskenteleviä. (Somasundaram 2013.)

Vaikka versionhallintahaaroja käytetään lähes jokaisessa versionhallintajärjestelmässä, pidetään Gitin versionhallintaa usein ylivoimaisena muihin verrattuna sen keveyden ja nopeuden takia. Git kannustaakin käyttäjää käyttämään versionhallintahaaroja hyödykseen usein (Chacon & Straub 2014). Yksi syy, miksi Gitiä suositellaan

yli muiden, on myös sen versionhallinta haarojen käyttö. Monissa muissa versionhallintajärjestelmissä ne esitellään edistyneen käyttäjän toimintoina, kun taas Gitissä ne kuuluvat jo alkeisiin. (Driessen 2010.)

2.3.3 Jatkuva integraatio ja jatkuva julkaisu

Jatkuva integrointi (Continuous integration) tarkoittaa sitä, että aina kun koodipohjaan tehdään muutoksia, käynnistetään testit sen laadun testaamiseksi. Kun saman koodin kanssa työskentelee useampi ohjelmistokehittäjä, jotka työstävät samaan aikaan sen eri versioita, on tärkeää tällä tavoin integraatiotesteillä säännöllisesti varmistaa, että muutokset sopivat yhteen. (Verona 2018.)

Jatkuva julkaisu ja jatkuva toimitus (Continuous delivery / Continuous deployment) yleensä jatkaa siitä, missä jatkuva integrointi on saanut testattua koodipohjan ja todettua sen toimivaksi, julkaistavaksi versioksi ja varmistaa, että ohjelmistosta on mahdollista julkaista toimiva versio napin painalluksella (Continuous delivery) tai jopa toimittaa sen tuotantoympäristöön (Continuous deployment). (Pittet n.d.)

Monet versionhallinta-alustat tarjoavat mahdollisuuden rakentaa alustaan sisäänrakennettuja ketjutuksia (pipelines), jotka tarkkailevat muutoksia tiettyyn versionhallintahaaraan ja havaitessaan sellaisen käynnistävät määritellyn ketjun komentoja, jotka suorittavat ohjelmakoodille automaattisia testejä, lähettävät näistä raportit ja näin määriteltynä onnistuneiden testien jälkeen julkaisevat ohjelmasta uuden version. Tämä mahdollistaa etenkin tietyissä sovellustyypeissä jatkuvan julkaisun tehokkaimmillaan, kun kehittäjät voivat julkaista uusia versioita tiheään ja pieninä inkrementteinä, kärsimättä kuitenkaan jatkuvan manuaalisen testaamisen ja julkaisemisen aiheuttamasta työnmäärästä. (Pittet n.d)

2.3.4 Sovelluksen elinkaaren hallinta

Sovelluksen elinkaaren hallinta (Application lifecycle management, ALM) käsittää sovelluksen hallinnon, kehityksen ja ylläpidon. Siihen kuuluu mm. vaatimusten hallinta,

ohjelmistoarkkitehtuuri, kehittäminen, testaaminen, ylläpito, muutoksen hallinta, jatkuva integrointi, projektin hallinta, käyttöönotto ja päivitysten hallinta. (Sovelluksen elinkaaren hallinta Microsoft Power Platformin kanssa – yleiskuvaus n.d.)

Jotkin ALM-sovelluksista integroituvat hyvin versionhallinnan kanssa ja antavat mahdollisuuden kytkeä yhteen mm. versionhallintaa, vaatimustenhallintaa, testiautomaatiota ja muita sovelluksen elinkaaren hallintaan kuuluvia osa-alueita. Esimerkiksi vaatimusmäärittelyn käyttäjätarinoita työstäessä voidaan luoda työnkulkuja (workflow), jonka avulla versionhallintaan luodaan repositorioon automaattisesti ominaisuushaara, kun ominaisuus otetaan työn alle.

2.3.5 Laadunvarmistus, katselmointi, ohjelmistotestaus

> *Ohjelma-arvioinnit, tarkastukset ja koodikatselmoinnit ovat testauksenhallinnassa tapahtumia, joiden yhteydessä tarkastetaan, että ohjelman komponentit ovat rakennettu oikein, ohjelmointikielen näkökulmasta järkeviä ratkaisuja käyttäen ja että koodin yleinen rakenne vastaa yleisesti hyväksyttyjä ja sovittuja ohjelmointikäytäntöjä. (Kasurinen 2013)*

Jotkin versionhallinta-alustat tarjoavat mahdollisuuden asettaa repositoriolle sääntöjä, jotka esimerkiksi edellyttävät muutosten katselmoinnin, ennen kuin ne voidaan yhdistää tiettyyn haaraan (Reviewing a pull request). Katselmoinnit mahdollistavat esimerkiksi sen, että kun uusi työntekijä työskentelee projektin parissa, muutokset täytyy tarkistaa, ennen kuin ne voidaan sulauttaa kehitys- tai päähaaraan. Tätä voidaan hyödyntää niin, että esimerkiksi yksi tai useampi kokeneempi ohjelmistokehittäjä tarkistaa, että koodin ulkoasu on yhdenmukaista ja ominaisuudet ovat toteutettu linjassa yrityksen käytäntöihin nähden. Voidaan myös esimerkiksi varmistaa, että koodia ei sulauteta, että ohjelmistotestaaja on suorittanut tarvittavat testit koodille.

3 Toteutus

3.1 Kehittämistyön menetelmä

Tutkimusmenetelmänä käytettiin työelämän kehittävää tutkimustoimintaa. Siinä lähestyttiin yleistä ajankohtaista ongelmaa ja etsittiin siihen yleishyödyllisiä ratkaisuja, hyödyntäen niitä toimeksiantajan prosessien kehittämisessä.

Tutkimus toteutettiin niin, että ensin tehtiin teoriapohjaista tutkimustyötä kriittisimpien kriteerien, kuten tietoturvan ja toimeksiantajan olemassa olevien käytäntöjen osalta, jotta mitkään käyttöönoton tai itse tutkimuksen vaiheet eivät aiheuttaisi haittaa toimeksiantajan liiketoiminnalle. Tämän jälkeen laadittiin suunnitelma versionhallinnan ja sen käyttöönotolle runko, jossa määriteltiin pääkohdat, joita ei olisi tarpeen muuttaa enää projektin aikana, kuten itse versionhallintajärjestelmä. Tämän jälkeen osa tutkimuksesta edettiin iteratiivisesti niin, että käyttöönoton eri osa-alueita voitiin suunnitella ja kokeilla käytännössä ja tämän jälkeen joko todeta toimiviksi, parannella tai muuttaa. Ydinajatuksena oli, että jokainen vaihe on suunniteltu niin huolellisesti, että se ei saa kasvattaa työntekijöiden työkuormaa tai projektien läpimenoaikoja, vaan vaikutuksen tulee joka tapauksessa olla niiden osalta positiivinen.

Tiedonkeruuseen käytettiin kirjallisuutta ja verkkolähteitä järjestelmiin, alustoihin, palveluntarjoajiin ja käyttöönottoon liittyen. Kriteereiden kartoittamiseen ja vertailemiseen eri ratkaisuehdotuksiin erityisesti tutkimuksen tekijän oppimaa ja kokemusta versionhallinnasta ja projektinhallinnasta, sekä toimeksiantajan henkilöstön ammattitaitoa ja tuntemusta olemassa olevista prosesseista.

3.2 Perehtyminen aiheeseen

3.2.1 Tietoturva

Toimeksiantajalle, niin kuin monelle muullekin yritys yksi tärkeimmistä asioista versionhallintaa käyttöönotettaessa on tietoturva. Järjestelmään talletettava lähdekoodi

on osa yrityksen tuotetta, tai tuote itsessään. Toisaalta tietomurrot- ja vuodot ovat myös usein haitallisia yrityksen imagolle.

Yksi tärkeimmistä asioista on, että työntekijöiden kirjautumistietoja ei pääse väärinkäyttämään. Monet hyökkäykset ovat johtuneet siitä, että yrityksen sisällä ei olla käytetty tarpeeksi vahvoja kirjautumismetodeja tai työntekijöiden tai projektien kirjautumistietoja on päässyt väärinkäyttäjien käsiin. Tämä on johtanut esimerkiksi kokonaisten repositorioiden ylikirjoittamisiin ja kiristykseen väärinkäyttäjien toimesta. Tämän takia yrityksen sisällä pitäisi vaatia mm. monivaiheisen tunnistautumisen ja vahvojen uniikkien salasanojen käyttämistä. (Robinson 2019.)

Käyttäjätunnusten lisäksi on tärkeää myös vahtia, että versionhallinta alustan asetukset on konfiguroitu oikein. Mikäli versionhallinta ylläpidetään omalla palvelimella, huolimaton määrittely voi mahdollistaa hyökkääjien pääsyn suoraan palvelimelle ilman käyttäjien tunnuksia (Nissan investigated source code exposure, says it plugged leak). Myös pilvipalvelua käytettäessä on oltava tarkkana, sillä mikäli repositorio esimerkiksi määritellään epähuomiossa julkiseksi, on kenellä vain pääsy lähdekoodeihin. Jotkut palveluntarjoajat tarjoavat turvatoimena mahdollisuuden sallia yhteyden repositorioihin vain tietystä IP-osoitteesta, jolloin voidaan rajata käyttö vain yrityksen turvalliseksi toteamiin yhteyksiin, kuten VPN-yhteyden kautta tai ainoastaan yrityksen omasta verkosta (Wilkes 2017).

Versionhallinnassa on tärkeää muistaa, että koska järjestelmän on tarkoitus pitää kirjaa versioiden muutoshistoriasta, ei silloin auta, että repositorioon vahingossa päässeet arkaluontoiset tiedot vain poistetaan. Ne jäävät tällöin myös versiohistoriaan, josta ne voi olla hyvin työlästä poistaa (Jackson 2020). Vaikka helposti ajatellaankin, että ongelma liittyy lähinnä julkisiin repositorioihin, katsotaan tällaisten tietojen tallentaminen versionhallintaan huonoksi käytännöksi. (Secrets detection for Application Security n.d.)

Ennen projektin siirtämistä versionhallintaan onkin siis tärkeä miettiä, mitä projektin tiedostoja ja tietoja ei haluta asettaa repositorioon jakoon ja miten ne järjestetään

niin, että ovat kuitenkin projektia työstävien saatavilla ja liitettävissä projektiin mahdollisimman vähällä vaivalla. Tiedostot, jotka sisältävät sensitiivistä informaatiota, voidaan jättää pois version hallinnasta esimerkiksi Git-versionhallinnassa gitignore-tiedostolla, jonka avulla voidaan määrittää mitkä tiedostot tai kansiot tallentuvat versionhallintaan ja mitkä eivät. Vaihtoehtona tiedostojen piilottamiselle versionhallinnasta on esimerkiksi hakea tiedot kuten tietokantayhteydet ja API-avaimet ympäristömuuttujista sen sijaan, että ne olisivat tekstimuodossa lähdekoodin joukossa. (Katz 2019.)

Kun organisaatiolla on projekteja, on loogista, ettei kaikki organisaatioon kuuluvatkaan työskentele kaikkien projektien parissa. Tällöin, erityisesti jos organisaatio on suuri tai projekteissa on erilaisia salassapito- tai turvaluokituksia, on organisaatiolla tarve rajata eri projektien näkyvyyksiä ja oikeuksia oman organisaationkin sisällä. Ohjelmistoyrityksen kannalta voi myös olla toivottavaa, ettei esimerkiksi kesätyöntekijä saa oikeuksia kaikkiin projekteihin. Versionhallinta-alustan ja sen sisältämien repositorioiden käyttäjäoikeudet on siis hyvä suunnitella ja kartoittaa etukäteen. (Ahlfors 2020.)

3.2.2 Paikallinen vai hajautettu

Versionhallinnassa on mahdollisuus käyttää paikallista, keskitettyä tai hajautettua versionhallintajärjestelmää. Paikallisessa järjestelmässä tiedostot ja niiden versiohistoria säilytetään paikallisella laitteella. Tämä on usein kaikista yksinkertaisin järjestelmä, mutta myös riskialttein, eikä se myöskään mahdollista ryhmän työskentelyä saman projektin parissa. (Somasundaram 2013.)

Keskitetty järjestelmä taas pitää tiedostoja palvelimella, joka on kaikkien projektin parissa työskentelevien saatavilla. Kun henkilö haluaa työskennellä yhden tai useamman tiedoston parissa, haetaan vain kyseisten tiedostojen uusimmat versiot. Kaikki muutokset tiedostoihin jaetaan muiden kanssa automaattisesti. (Somasundaram 2013.)

Hajautettu järjestelmä on eräänlainen yhdistelmä paikallista ja keskitettyä järjestelmää. Siinä projektit ovat sekä keskitetyllä palvelimella, että kehittäjän omalla laitteella. Hajautetulla järjestelmällä on useita etuja keskitettyyn nähden. Jotta käyttäjällä olisi keskitetyssä järjestelmässä pääsy tiedostoihin, tulee tällä olla yhteys palvelimeen, jossa tiedostot sijaitsevat. Hajautetulla järjestelmällä jatkuvaa yhteyttä ei tarvita, sillä käyttäjän koneelle haetaan kopio koko repositoriosta, jolloin yhteyttä palvelimeen tarvitaan vain silloin, kun haetaan tieto viimeisimmistä muiden tekemistä muutoksista, tai tallennetaan omat muutokset päärepositorioon muiden haettavaksi. Koska hajautetussa versionhallinnassa haetaan käyttäjän laitteelle koko repositorio, eli projektin kaikkien tiedostojen kaikki versiot, toimii se myös itsessään varmuuskopiona projektille. Keskitetyllä palvelimella kaikki tiedostot ovat yhdessä paikassa, kun taas hajautetussa järjestelmässä kaikilla kehittäjillä on kopio projektista koneella. (Somasundaram 2013.)

Keskitetyllä järjestelmällä on joitakin etuja hajautettuun verrattuna. Yksi niistä ajatellaan yleisesti olevan sen käyttöönoton ja oppimisen helppous, sillä siinä on vähemmän komentoja. Toinen on sen suorituskyky projekteissa, joissa on suuria tiedostoja. Hajautetuissa järjestelmissä repositorio haetaan kokonaisuudessaan käyttäjän koneelle, kun taas keskitetyssä järjestelmässä haetaan vain tiedosto, jota muokataan. Useissa ohjelmistoprojekteissa kuitenkin suurimmat tiedostot ovat binääritiedostoja, jotka generoidaan ohjelmakoodista, jolloin tämä ei ole merkittävä etu. Tietyissä tapauksissa käyttöoikeuksien hallinnoiminen voi olla helpompaa paikallisen järjestelmän avulla, mutta se taas riippuu yrityksen ratkaisuista käyttäjänhallinnan suhteen yleisesti ottaen. (Upadhyay 2019.)

3.2.3 Oma palvelin vai pilvipalvelu

Monet versionhallinta-alustat ovat saatavilla joko pilvipalveluna, tai asennettavan ohjelmana omalle palvelimelle. Molemmissa ratkaisuissa on hyvät ja huonot puolensa, jotka riippuvat paljon organisaation rakenteesta ja resursseista.

Yksi suurimmista eduista palvelun ylläpitämisestä itse on se, että yritys saa itse päättää mihin palvelimet ja täten tiedostot ovat fyysisesti sijoitettu. Samoin ne ovat näin

suoraan yrityksen omassa hallinnassa, eikä siihen liity välikäsiä niin kuin pilvipalvelun ylläpitäjät. Siinä missä tietoturvan vertailussa on useampi eri vaikuttava tekijä, on ainakin itse ylläpidetyn palvelimen tietoturvan auditointi varmempaa. Toisaalta omien palvelimien tietoturvan määrittäminen, ylläpito ja auditointi aiheuttaa huomattavasti lisää työkuormaa ja tukiprosesseja yritykselle, sekä luo usein tarpeen tietoturvaan erikoistuneelle henkilöstölle. (Ahlfors 2020.)

Siinä missä monesti ajatellaan pilvipalvelujen olevan omaa palvelinta kalliimpi tai vähemmän turvallinen vaihtoehto, voi silti usein olla päinvastoin. Kun versionhallintapalvelin on ylläpidetty omissa tiloissa, tiedetään missä koodit sijaitsevat ja teoriassa kuka niihin pääsee käsiksi. Oma palvelin vaatii kuitenkin sekä tietoturvan tietotaitoa, että myös ylläpitoa, joka sisältää mm. päivitykset ja varmuuskopiointi (On-Premise Source Code Management). Oman palvelimen kustannuksia on mm. ohjelmistolisenssit, laitehankinnat, tilakustannukset, sähkönkulutus sekä käytetty työ palvelimen asentamiseen ja ylläpitoon. Huolimattomuus ylläpidossa voi johtaa myös vakaviin tietovuotoihin (ks. esim. Nissan investigated source code exposure 2021, says it plugged leak, Cimpanu 2020).

Kustannusten ero ylläpidetyn palvelimen ja pilvipalvelujen välillä vaihtelee hyvin paljon palveluntarjoajan mukaan. Tässä tulee kuitenkin ottaa huomioon se, että siinä missä ohjelmistolisenssit ovat usein suoraan vertailtavissa näiden vaihtoehtojen välillä, tulee kustannuksia myös omalle palvelimelle muun muassa laitehankinnoista, virrankulutuksesta, ylläpitoon käytettävistä työtunneista, kahdennuksista ja varmuuskopioinneista.

Yrityksen projektien ollessa henkilöstömäärältään pieniä, ovat pilvipalveluna tarjottavat versionhallinta-alustat käyttäjämäärään perustuvan hinnoittelun takia varsin edullisia ja tiettyyn pisteeseen asti jopa ilmaisia.

3.2.4 Palveluntarjoajan valitseminen

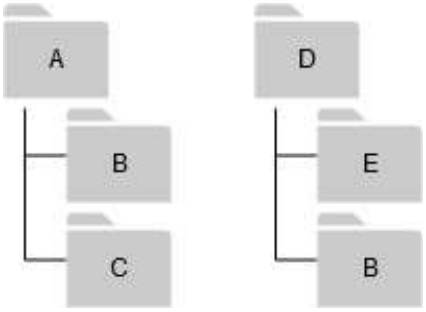
Versionhallinnan ylläpitoon on monia eri palveluntarjoajia, joista jotkut osa on erikoistunut vain avoimen lähdekoodin repositorioiden ylläpitämiseen. Yrityksen luottaessa oman tuotteensa, eli ohjelmiston lähdekoodin jonkin palveluntarjoajan ylläpitämäksi, on usein tärkeää, että kyseisen palveluntarjoaja on tunnettu ja siitä löytyy tarpeeksi konkreettista käyttäjäkokemusta niin turvallisuudesta kuin käytettävyydestäkin. Useissa eri yrityksen edustajille tehdyissä haastatteluissa, sekä verkossa tehdyissä kyselyissä erottui muutama suosittu palveluntarjoaja: Azure DevOps, Bitbucket, GitHub ja GitLab. Vertailu suoritettiin siis lähinnä näiden palveluntarjoajien välillä.

Palveluntarjoajaa valitessa yksi olennainen kriteeri on luonnollisesti hinta. Hinnat vaihtelevat usein sekä käyttäjien määrän, että palveluun kuuluvien ominaisuuksien mukaan. Useat palveluntarjoajat tarjoavat palvelusta ilmaisversiota, johon kuuluu yleensä rajallinen määrä käyttäjiä sekä ominaisuuksia.

Yksi valintakriteeri on alustan integroitavuus muihin yrityksen käyttämiin työkaluihin. Versionhallinta on itse ohjelmistokehityksen lisäksi oleellinen osa muun muassa sovelluksen elinkaaren hallintaa, jatkuvaa integrointia, jatkuvaa julkaisua, vianhallintaa ja ohjelmistotestausta. Tämän takia on yrityksen prosessien kannalta kannattavaa kartoittaa, miten nämä eri työkalut integroituvat valittavan versionhallinta-alustan kanssa.

3.2.5 Projektien rakenne

Projektien siirtämisessä versionhallinnan alustaan on hyvä miettiä niiden suhdetta toiseensa ja mahdollisia sisäkkyyksiä ja viittauksia niiden kesken. Kuvitellaan esimerkiksi tilanne, jossa meillä on projektit A, B, C, D ja E. Projektit B ja C ovat projektin A komponentteja, kun taas E on D:n komponentti, mutta tämän lisäksi D käyttää myös A:n komponenttia B (Kuvio 1).

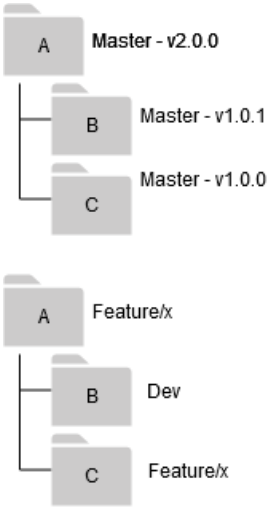


Kuvio 1 – Projektit ja niiden kirjastot sisäkkäisessä kansiorakenteessa

Sisäkkäisiä projekteja siirtäessä versionhallintaan tulee miettiä, miten niistä muodostaa repositorioita, sekä miten hoitaa niiden viittaukset toisiinsa.

Mikäli meillä olisi vain projekti A ja sen alaprojektit, voitaisiin luoda yksi repositorio, joka sisältäisi koko kyseisen kansiorakenteen. Tässä tapauksessa kuitenkin projekti B jouduttaisiin lisäämään myös D:n repositorioon, jolloin muutokset B: repositorioon pitäisi tehdä kahteen paikkaan, tai muuten projektilla olisi vaarana elää omaa elämäänsä molemmissa repositorioissa.

Toinen vaihtoehto on muodostaa jokaisesta komponentista oma repositorionsa. Näin projektit voidaan liittää toistensa alle kirjastoina moduuleilla (submodules), jolloin useampaa repositoriota voidaan käyttää toistensa sisällä, mutta silti pitää niiden muutokset erillään toisistaan, omissa repositorioissaan. Moduuleina kirjastot ovat viittauksia toisen repositorion tiettyyn versioon (Kuvio 2). Moduulien avulla projektin käyttämiin kirjastoihin on helpompi tehdä muutoksia kuin paketinhallintatyökalujen avulla ladattaviin, mutta niiden hallinta on monimutkaisempaa. Tämän takia moduulien käytöstä on enemmän hyötyä, kun projektissa käytetään kirjastoja, joihin halutaan tehdä aktiivisesti muutoksia, jotka halutaan tuoda myös muiden projektien käytettäväksi. (Chacon & Straub 2014.)



Kuvio 2 – Projektit ja niiden kirjastot repositoriomoduuleina

Projektin rakennetta suunniteltaessa on myös hyvä miettiä sen vaikutusta projektin versionumerokäytäntöihin. Mikäli projektit ovat kaikki yhdessä repositoriossa, täytyy käytännössä kaikkien projektin osien versionumeroiden päivittyä, kun sen yhtä osaa muutetaan. Toisaalta, jos osia ei käytetä muissa projekteissa, tämä on ehkä luontevaakin. Mikäli taas projektin osat on jaettu moduuleihin, on tällöin helpompi päivittää yhtä projektin komponenttia kerrallaan, mutta kaikkien projektien yhtenäinen versionumeron korotus on työläämpää. (Ahlfors 2021.)

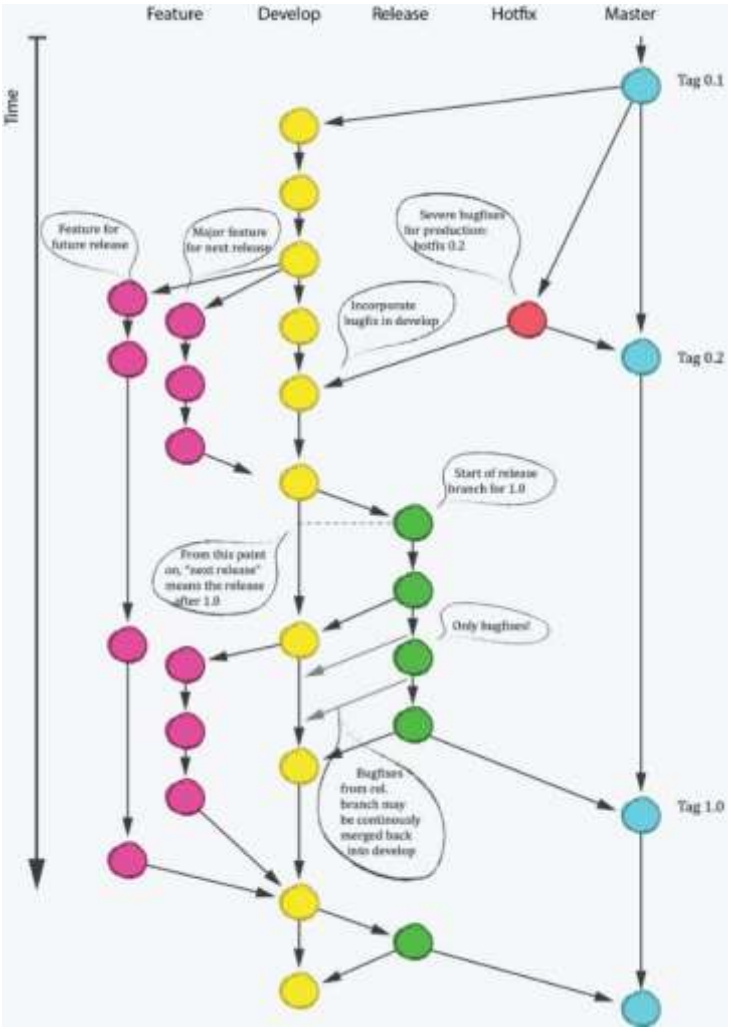
3.2.6 Versionhallinnan strategiat

Branching Strategy on ohjeistus, joka määrittää muun muassa koska repositorion haaroja luodaan ja miten nimetään ja mihin niitä käytetään (Verona, J. 2018). Tämä mahdollistaa useamman ominaisuuden samanaikaisen kehittämisen toisistaan erillään, joka taas mahdollistaa esimerkiksi valittujen ominaisuuksien julkaisemisen ilman, että julkaisuun päätyy mukaan esimerkiksi keskeneräistä tai testaamatonta sovelluskoodia, tai ominaisuuksia, jotka halutaan julkaista vasta myöhemmin (Driessen 2010).

Eräs suosittu malli on Vincent Driessenin esittelemä ”git-flow” – malli (Kuvio 3 – Git-flow strategia, mukailtu Vicent Driessenin mallista (Driessen 2010). Git-flow suosittu

strategia ja vaikka se onkin monille liian monimutkainen, sitä voi tarvittaessa yksinkertaistaa (Verona 2018). Git-flow sopii esimerkiksi ohjelmistoille, joista täytyy ylläpitää useampaa eri versiota (Driessen. 2010).

Git-flow mallissa repositoriossa on kaksi pysyvää haaraa, master (päähaara) ja develop (kehityshaara). Master-haaran viimeisin versio lähdekoodista on aina julkaisukelpoinen. Develop-haara taas vastaa viimeisimpiä muutoksia, jotka ovat tulossa seuraavaan julkaisuun. Develop-haara toimii integraatiohaarana. Aina kun uusi koodi on valmista ja valmis julkaistavaksi, se sulautetaan takaisin master-haaraan. (Driessen 2010.)



Kuvio 3 – Git-flow strategia, mukailtu Vicent Driessenin mallista

Master- ja develop-haarojen katsotaan olevan repositorion päähaarat ja niiden lisäksi siihen kuuluu myös tukihaaroja, joiden tehtävä on muun muassa tukea tiiminjäsenten samanaikaista kehitystyötä ja auttaa seuraamaan uusia ominaisuuksia ja nopeuttaa viankorjausta tuotannossa ilmeneviin ongelmiin. Tähän käytetään Feature- Release- ja Hotfix- haaroja (ominaisuus-, julkaisu- ja korjaushaara). Feature-haaraa käytetään uuden ominaisuuden kehittämiseen. Feature-haaran elinikä on sen aikaa, että uusi ominaisuus saadaan kehitettyä valmiiksi erillään muusta koodista ja sulautetaan sen jälkeen Develop-haaraan. Release-haaran tarkoitus on auttaa uuden julkaisun valmistelemisessa. Kun Develop-haara on *ominaisuuksiltaan* valmis julkaistavaksi, siitä eriytetään uusi Release-haara, joka nimetään julkaistavan versionumeron mukaan. Nyt tulevaa versiota voidaan testata ja korjailla Release-haarassa ennen julkaisua, kun taas uudet samaan aikaan kehitettävät ominaisuudet sulautetaan edelleen Develop-haaraan. Hotfix-haarojen tarkoitus on korjata tuotannossa ilmaantuvia ongelmia nopeasti. Hotfix-haara eriytetään viimeisimmästä tuotantojulkaisusta, eli master-haarasta ja kun korjaus on valmis, se sulautetaan sinne takaisin. Näin korjaukset saadaan tehtyä ilman, että kehityksen alla olevaa koodia sotkeentuisi mukaan korjaukseen. (Driessen 2010.)

Koska Git-flow strategiassa julkaisuvalmis koodi sulautetaan master-haaraan ja master-haaraan sulautettava koodi on aina julkaisukelpoista, voidaan päähaaran uusinta versiota käytännössä oletuksena pitää uutena julkaisuversiona. Tämän perusteella voidaan mm. automatisoida haaraan mm. automaattisen julkaisun toimintoja (Driessen 2010.)

Toinen suosittu strategia on GitHub Flow. Se on suunniteltu paremmin projekteille, jotka käyttävät jatkuvaa julkaisua hyvin usein, kuten päivittäisellä tasolla. (Git Essentials) GitHub Flow sopii esimerkiksi verkkosovelluksiin, joihin sovelletaan usein jatkuvaa julkaisua, eikä niitä palauteta vanhoihin versioihin (engl. rollback), eikä niistä yleensä tarvitse ylläpitää useampaa eri versiota (Driessen 2010).

GitHub Flow – strategiassa ominaisuushaarat otetaan suoraan päähaarasta ja sulautetaan takaisin päähaaraan, jonka jälkeen tehdään välittömästi uusi julkaisu. Siinä työnkulussa jää siis pois develop- ja release- haarat kokonaan pois. (GitHub Flow n.d.)

3.2.7 Versionumerointi

Versionumerointi on tärkeä osa versionhallintaa. Versionhallintaan tallennetaan tavallisesti säännöllisesti tilannekuvia kehitettävästä ohjelmasta, joka tarkoittaa, että toimivien, asennettavien ohjelmaversioiden välissä on keskeneräisiä versioita. Tästä syystä on tapana merkitä asennettavat versiot tunnisteilla (tag), jotka vastaavat versionumeroita. Tietyissä sovelluksissa voi olla myös samasta ohjelmistosta samanaikaisesti käytössä useampia eri versioita, jonka versiot täytyy voida erottaa toisistaan (Driessen 2010). Veronan mukaan (Verona 2018) käytäntö on merkitä versio nelinumeroisella tunnisteella, jossa numerot symboloivat seuraavia asioita kyseisestä versiosta:

1. Ensimmäinen numero on isoille muutoksille, mahdollisesti taaksepäin yhteensopimattomia
2. Toinen numero pienemmille muutoksille, taaksepäin yhteensopiville
3. Bugikorjauksille, ei uusia ominaisuuksia
4. Build – numero, joka kertoo, kuinka mones koontiversio on menossa. Käytetään yleensä testauksessa.

3.3 Yrityksen tarpeiden ja prosessien kartoitus

3.3.1 Versionhallintajärjestelmä

Toimeksiantajalle valittiin versionhallintajärjestelmäksi Git, sillä sitä pidetään nykyään suorituskykyisenä, selkeänä ja eheyden kannalta varmana järjestelmänä (Somasundaram 2013). Hajautetun järjestelmän eduiksi koettiin riippumattomuus yhteydestä palvelimeen ja toisaalta repositorioiden kloonien toimimisen varmuuskopioina. Git valittiin muiden hajautettujen järjestelmien, kuten Mercurialin sijaan sen suosion vuoksi.

3.3.2 Asiakasympäristöt

Versionhallintaa käyttöönottaessa täytyy arvioida projektikohtaisesti asiakaskohtaiset rajoitukset. Rajoituksia voi kohdistua esimerkiksi versionhallinnan palveluntarjoajan maantieteelliseen sijaintiin ja joitain lähdekoodeja ei voida säilyttää sopimus- tai

lakiteknisistä syistä pilvessä laisinkaan. Koska toimeksiantajan asiakkuuksiin kuuluu erilaisia toimijoita, on versionhallinta-alustan sopivuutta tarkastella jokaisen projektin kohdalla erikseen.

Asiakasympäristö ja sovelluksen käyttötarkoitus voi vaikuttaa myös tiettyihin strategioihin, kuten jatkuvaan julkaisuun. Koska toimeksiantaja toteuttaa paljon ohjelmistoja teollisuusympäristöön ja ne toimivat usein yhdessä automaation kanssa, ei jatkuvaa julkaisua voi välttämättä toteuttaa nopeatempoisella syklillä ja käyttöönottoa ei voida automatisoida, sillä se vaatii mm. manuaalisia varotoimenpiteitä ja valvontaa.

3.3.3 Organisaation henkilöstö

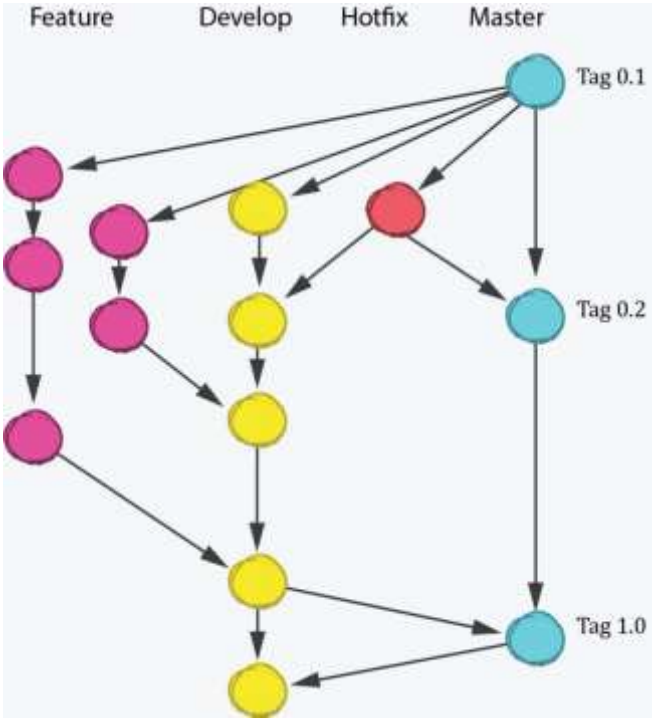
Organisaation henkilöstön koko on hyvä ottaa huomioon versionhallinta-alustaa miettiessä. Kehitystiimin koko voi vaikuttaa hyvin ratkaisevasti eri lisenssimaksuihin sekä eri palveluntarjoajien välillä, että myös pilvipalvelun ja oman palvelimen välillä. Oman palvelimen aiheuttaman vaivan ja lisäkustannuksien kannalta on myös ero siinä, onko organisaatiolla ennestään käytössä omia palvelimia ja henkilöstöä, joka ylläpitää niitä.

3.3.4 Projektien rakenne

Projektien komponentit, jotka olivat suurimmaksi osaksi NET – framework – projekteja, päädyttiin jakamaan omiin repositorioihinsa ja niiden keskinäiset riippuvuudet ratkaistiin alimoduuleilla (Git submodule), eli komponentit asetettiin viittaamaan toisten repositorioiden tiettyihin versioihin. Tässä nähtiin yhtenä etuna se, että projektien testaaminen ja vianmääritys oli helpompaa, kun komponentit olivat liitetty kokonaisuuteen lähdekooditasolla. Toiseksi projekteissa käytettyjä kirjastoja käytettiin yleensä muissakin sisäisissä projekteissa ja niitä kehitettiin usein jatkuvasti ja moduulien käyttö mahdollisti tämän kehitys- ja testaustyön lennosta pääprojektin yhteydessä.

3.3.5 Strategioiden yhteensovitus prosesseihin ja projektien työnkulkuun

Koska teolliseen ympäristöön suunniteltujen, automaation kanssa yhteen liitettyjen sovelluksien käyttöönotto vaatii usein valvontaa, varotoimenpiteitä ja tietyssä tapauksessa käyttökatkoksia, ei niissä voi soveltaa jatkuvan julkaisun muotoa, jossa sovellusten käyttöönotto automatisoitaisiin aktivoitumaan itsestään päivityksestä master-haaraan. Tämän takia git-flow vaikutti ensisijaisesti sopivalta vaihtoehdolta, mutta siitä päätettiin tehdä mukailtu versio (Kuvio 4 – Git-flowsta muunnettu haarastrategia, ensimmäinen luonnos. Koska ohjelmistokehityksen osasto oli ensinnäkin varsin pieni, päätettiin kokeilla jättää release-haara pois strategiasta, sillä se lisäisi ylimääräisen työvaiheen, mutta sen ei nähty tuovan valtavasti lisäarvoa. Release-haaran tarkoitus nimittäin on tarkoitus lukita Develop-haaran ominaisuudet ja mahdollistaa uusien ominaisuuksien lisäys kehityshaaraan, kun julkaisuhaaraan tulisi enää korjauksia. Koska toimeksiantajan tapauksessa kuitenkin ohjelmiston julkaisu tapahtuu yleensä paikan päällä ja viimeisin järjestelmätestaus tapahtuu yleensä yhden ohjelmistosuunnittelijan toimesta, voi julkaisuhaara yhtä hyvin sijaita kyseisen henkilön koneella paikallisesti.



Kuvio 4 – Git-flowsta muunnettu haarastrategia, ensimmäinen luonnos

Strategiaan päätettiin tehdä myös toinen muutos. Alkuperäisessä Git-flow – mallissa uusi Feature-haara aloitetaan aina Develop-haarasta (Driessen 2010). Tässä on etuna se, että kun haara aloitetaan kehityshaarasta ja sulautetaan valmistuessaan takaisin, on uudessa ominaisuushaarassa aina muut valmistuneet muutokset jo mukana, mikä taas vähentää konfliktien määrää sulauttaessa haaran takaisin. Toimeksiantajalla tässä nähtiin kuitenkin mahdollinen haaste siinä, että koska yhdestä projektista oli erilaisia vaatimuksia eri asiakkailla ja tietyt ominaisuudet toteutettiin jo hyvissä ajoin, vaikka ei välttämättä tiedetty missä versiossa ne tultaisiin julkaisemaan, kaivattiin hallintaa siihen, mitkä jo valmiit ominaisuudet julkaistaisiin missäkin vaiheessa. Tätä yritettiin lähestyä strategialla, jossa uusi ominaisuus haarautettaisiin Master-haarasta, ja sulauttaa kehityshaaraan vasta lähempänä julkaisua, jolloin olisi mahdollista julkaista se halutessaan vaikkapa ainoana ominaisuutena seuraavassa julkaisussa ja seuraavassa versiossa julkaistavat ominaisuudet voitaisiin päättää lähempänä julkaisua, kun ne eivät sisältäisi muutoksia muista ominaisuushaaroista.

Muuten strategia seuraisi git-flow – mallia: Master- eli päähaaraan tulisi ainoastaan julkaisuversioita. Kehitettävät ominaisuudet ja bugikorjaukset tulisivat jokainen omaan haaraansa. Sen lisäksi ominaisuuksien testausta varten olisi oma haaransa, QA-haara (Quality Assurance, laadunvarmistus), joka vastaisi git-flow:n Develop-haaraa. QA-haarassa on aina Master-haaran viimeisimmät muutokset, jotta voidaan testata myös uusien päivitysten vaikutus vanhoihin ominaisuuksiin. Kun uusi ominaisuus on valmis ja sen toimivuus on testattu sellaisenaan omassa haarassaan, se sulautetaan QA-haaraan integraatiotestausta varten. Kun kaikki seuraavaan julkaisuun valitut ominaisuudet on sulautettu yhteen ja ohjelmistolle on suoritettu vaadittavat integraatio- ja regressiotestit, se voidaan sulauttaa Master-haaraan uudeksi julkaisuversioksi. Tilanteessa, jossa ohjelmistolle tarvitsee tehdä pikainen korjaus ”hotfix”, korjaushaara sulautettaisiin suoraan master-haaraan.

4 Käyttöönotto

4.1 Käyttöönoton metodi

Käyttöönottovaiheessa otettiin tutkimustulosten perusteella ja niiden aikana tehtyä dokumentointia noudattaen käyttöön versionhallinta Bitbucketin pilviympäristöön, sekä sitä tukemaan Atlassianin Jira ensisijaisesti tehtävän- ja virheidenhallintaan. Käyttöönottoa työstettiin pikkuhiljaa normaalien prosessien ohella. Projekteja siirrettiin versionhallintaan yksi kerrallaan ja niihin otettiin aluksi yksi ohjelmistokehittäjä työskentelemään niiden parissa, kunnes projektin parissa työskentelevien määrää lisättiin yksi kerrallaan. Näin käyttöä kerettiin opettelemaan rauhassa, ilman että prosessit kärsivät.

4.2 Pilvipalvelu pienelle tiimille

Toimeksiantajalla päädyttiin valitsemaan palveluntarjoajan ylläpitämä pilvipalvelu. Haastatellessa toimeksiantajan henkilöstöä, sekä useampaa eri ohjelmistoalan yrityksen edustajaa saatiin yksimielinen mielipide siitä, että muutaman ohjelmistokehittäjän kokoiselle tiimille on sekä edullisempaa, että usein turvallisempaa valita palveluntarjoaman ylläpitämä palvelin. (Ahlfors 2021.)

Omalle palvelimelle asennettavista ohjelmistoista verrattiin muutamaa suosituinta versionhallintapalvelinsovelluksia, jotka pieniotantaisen haastattelun ja useiden verkkokyselyiden mukanaan olivat Microsoftin, Atlassianin ja GitLabin tarjoamat ohjelmistot. Näistä GitLab osoittautui erityisen käytetyksi, osittain siksi, että sen saa asennettua omalle palvelimelle ilmaiseksi. Muut ohjelmistot olivatkin turhan kalliita siihen nähden, että useimmat ohjelmistot pilvipalveluina olivat ilmaisia käyttää sen kokoisille kehitystiimeille, kun toimeksiantajalla tutkimuksen hetkenä.

Siinä missä lisenssimaksujen puolesta GitLabin palvelinohjelmisto pärjäsi vertailussa, olisi sen asentamiseen ja ylläpitämiseen vaadittujen kustannusten katsottu olevan

liian korkeat organisaation kokoon nähden, sillä sen ylläpitäminen olisi vaatinut laitehankintoja, tilajärjestelyjä sekä suhteettoman paljon työtunteja henkilöstön kokoon nähden. Myös tietoturvaa ei pidetty tarpeeksi vahvana kyseisessä ratkaisussa, sillä ohjelmisto vaatisi Linux-alustan, eikä yrityksen käytössä käyttöönoton hetkellä ollut riittävästi Linux-osaamista palvelimen jatkuvaan ylläpitämiseen.

Huomioitavaa on, että kaikkia toimeksiantajan projekteista ei voitu siirtää käyttöönotettavaan versionhallintajärjestelmään, sillä jotkin asiakkaista asettavat hyvin tiukkoja säädöksiä sille, missä ohjelmistokoodeja maantieteellisesti säilytetään ja ettei edes teoriassa kolmannella osapuolella saa olla mahdollisuutta päästä käsiksi niihin. Monilla yrityksillä on tällaisia rajoitteita, jotka kannattaa käydä projektikohtaisesti läpi, ennen kuin versionhallinnan sijaintia mietitään.

4.3 Palveluntarjoajan valinta

Palveluntarjoajaa tarkasteltiin ensisijaisesti tietoturvan, hinnan, käytettävyyden ja integroitavuuden näkökulmasta. Vertailtavien palveluntarjoajien suodattamiseksi haastateltiin muutamia ohjelmistoalan yrityksissä työskenteleviä työntekijöitä, sekä tutkittiin verkkoartikkeleissa ja -kyselyissä esiintyviä nimiä. Syvempään tarkasteluun valikoituivat Microsoft Azure DevOps, Atlassian Bitbucket, GitLab ja GitHub.

Azure DevOps tarjosi monipuolisia palveluita versionhallintaan liitettäväksi, kuten Kanban-taulua työnkulun, tehtävien ja bugien hallintaan, sekä testisuunnittelutyökalua testitapausten määrittelyyn. DevOpsissa saatavilla oli myös Pipelines-ominaisuus, joka mahdollistaa mm. automaattisten testien ajamisen sekä julkaisujen hallintaa. Vaikka liitettävät toiminnot vaikuttivat monipuolisilta, oli ilmainen käyttö jokseenkin rajattua, jonka jälkeen käyttömaksut nousivat nopeasti. Suurin puute DevOps-palvelussa oli kuitenkin monivaiheinen tunnistautuminen. Microsoftin sivut antoivat ymmärtää, että monivaiheinen tunnistautuminen onnistuisi vain, mikäli kaikilla käyttäjillä olisi käytössään Office365-tili, joka oli verrattain suuri kustannus käyttötarkoitukseensa verrattuna. Tämä tarkistettiin myös ottamalla yhteyttä suoraan Microsoftin asiakaspalveluun, josta asia vahvistettiin. Koska monivaiheinen tunnistautuminen oli

yksi tärkeimmistä prioriteeteista, DevOps hylättiin tässä kohtaa. (What features and services do I get with Azure DevOps? 2020.)

Myös GitLab tarjosi paljon erilaisia ominaisuuksia versionhallinnan rinnalle, sekä tietoturva-asetuksia käyttäjähallintaan. GitLab tarjosi muun muassa mahdollisuuden pakottaa kaksivaiheisen tunnistautumisen kaikille ryhmän jäsenille, mikä olisi suuri etu isommassa organisaatiossa, jossa käyttäjien tietoturva-asetusten manuaalinen varmistus ja auditointi olisi työläämpää. (All GitLab Features.) Tutkimuksen ajankohtana GitLab ilmoitti, että se ei ylläpidä palvelujaan itse omassa konesalissaan, vaan ne sijaitsevat Google Cloud Platformissa. (Newdigage 2018.)

GitHubia on pidetty yleisesti avoimen lähdekoodin alustana. Viime aikoina se oli kuitenkin lisännyt tukea myös yksityisille repositorioille, esimerkiksi tarjoamalla yksityiset repositoriot tiimeille ilmaiseksi (Friedman 2020). GitHub tuki myös organisaatioympäristöjä, jossa repositorioita voidaan omistaa ja hallita paremmin ryhmänä yksilön sijaan (Neath, 2010). GitHubin siirtyminen Microsoftin omistukseen oli kuitenkin verrattain tuore ja haastattelujen mukaan monet ohjelmisto-organisaatiot mieluiten odottavat ensin ja katsovat, miten omistajanvaihdos vaikuttaa palvelun laatuun ja tasaisuuteen (Microsoft to acquire GitHub for $7.5 billion, 2018). Myös GitHubin maine avoimen lähdekoodin alustana vaikutti tutkimusta tehdessä vielä monen haastatellun mielipiteeseen, mikä otettiin myös huomioon, koska yleinen maine ja vaikutelma alalla vaikuttaa väistämättä myös valintaa tehdessä. Haastattelujen, tutkimuksen yhteydessä tehdyn vertailun ja useamman verkkoartikkelin mielestä GitHubin integroitavuus tai helppokäyttöisyys ei ollut yhtä hyvä kuin esimerkiksi Bitbucketin (ks. esim. Li 2021).

Bitbucket valittiin versionhallinnan alustaksi seuraavin kriteerein. Se oli monen kotimaisen ohjelmistotalon suosittelema, sekä myös yksi suosituimmista alustoista kaikissa vastaan tulleissa verkkokyselyissä. Bitbucket oli ilmainen alle kuuden käyttäjän kehitystiimeille, mikä helpotti kynnystä käyttää alustaa (Plans and pricing). Bitbucket tarjosi mahdollisuuden käyttää kaksivaiheista tunnistautumista, sekä maksullisen lisenssin myötä IP-osoite – rajausta. Atlassian tarjosi myös mahdollisuuden perustaa Atlassian-organisaation. Se mahdollistaa käyttäjätunnusten linkittämisen yrityksen

verkkotunnuksen avulla samaan organisaatioon. Tämän ansiosta organisaation pääkäyttäjä voi hallita yrityksen käyttäjien tilejä kootusti, esimerkiksi asettamalla tiettyjä tietoturvakäytänteitä. (What is an Atlassian organization? n.d.)

4.4 Käyttäjäoikeuksien hallinta

Pienestä kehitystiimistä huolimatta käyttäjäryhmät ja -oikeudet mietittiin pidemmällä tähtäimellä ottaen huomioon kasvavan kehitystiimin. Versionhallinta-alustaan luotiin oma käyttäjäryhmä, jolla on korotetut oikeudet hallita projekteja ja repositorioita versionhallinnan sisällä. Muutaman kehittäjän ryhmälle tämä voi vaikuttaa ylimitoitetulta toimenpiteeltä. On kuitenkin hyvä, että kun kehitystiimi kasvaa ja otetaan esimerkiksi kesätyöntekijöitä, ei jokaisella uudella käyttäjällä ole oikeuksia ja näkyvyyttä kaikkiin toimintoihin ja asetuksiin. (Ahlfors 2021.)

Repositorioiden omistusoikeuden kannalta harkittiin erillisen päätilin luomista, jonka omistukseen repositoriot laitettaisiin, sillä Bitbucketissa repositoriot voivat olla vain henkilön, ei organisaation omistuksessa. Näin voitaisiin estää yksittäistä käyttäjää poistamasta repositoriota. Ajatuksesta kuitenkin toistaiseksi luovuttiin sen aiheuttaman lisätyömäärän vuoksi, sillä vaikka käyttäjä poistaisikin projektin päärepositorion, on Gitin yksi vahvuus juuri siinä, että jokainen kloonattu repositorio toimii projektin varmuuskopiona.

Haaroille luotiin myös joitain käyttöoikeusrajoituksia. Projekteista, jotka olivat esimerkiksi muita kriittisempiä, estettiin muutosten tekeminen suoraan master-haaraan. Näin vältyttiin vahingoilta, joita muuten olisi epähuomiossa tapahtunut ohjelmistokehittäjän unohtaessa vaihtaa haaraa, mikä taas olisi voinut riskeerata masterhaaran eheyden. Sen sijaan projektista riippuen päähaaran muuttaminen sallittiin ainoastaan toisesta haarasta sulauttamalla (merge), mikä toisissa projektissa kovennettiin vielä säännöllä, että ainoastaan tiettyyn ryhmään kuuluvat voisivat sulauttaa haaroja, kun joku tiimin jäsenistä asettaisi sille ensin pyynnön (pull request). Näin esimerkiksi voidaan varmistaa, että uusi työntekijä ei voi tehdä muutoksia päähaaraan, ennen kuin muutokset on katselmoitu jonkun kokeneemman toimesta.

4.5 Standardit ja dokumentointi

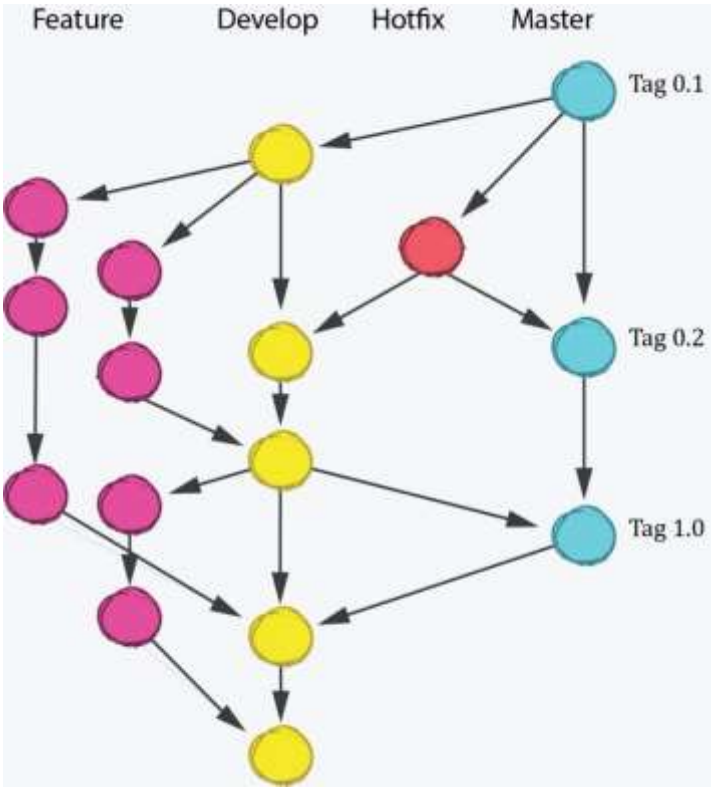
Kun yritykselle otetaan käyttöön versionhallinta, liittyy siihen monia ei työvaiheita ja muita käytänteitä. Tehokkuuden ja tietoturvan lisäämiseksi ja vaihtelevuuden vähentämiseksi prosesseissa on tärkeää standardisoida ja dokumentoida niitä mahdollisimman paljon. Kun yksinkertaisimmatkin vaiheet tehdään aina samalla tavalla, voidaan vähentää virheiden määrää, sekä helpottaa niiden vianmääritystä, kun niitä ilmenee.

Oli henkilökunta kokenutta tai vasta-alkajia versionhallinnan käytön kanssa, on silti parempi laatia yhteiset säännöt, miten ja milloin tiettyjä toimintoja käytetään. Esimerkiksi Git rebase voi olla monissa tilanteissa hyvin hyödyllinen ja tehokas komento, mutta monesti suositellaan, ettei sitä käytettäisi, jotta varmistettaisiin projektin yhtenäinen historia. (Mørken 2017). Toisaalta siinä missä rebasen katsotaan toisaalta muuttavan projektin historiaa, sen toisaalta myös katsotaan siistivän sitä luettavammaksi (Chacon & Straub 2014). On siis tärkeää, että käyttäjät ymmärtävät milloin eri komentoja käytetään ja mitä niillä saavutetaan. Käyttöönotossa koettiin, että on varmempaa ensin totuttautua perusteisiin ja priorisoida projektien ja niiden historian eheys ja hioa yksityiskohtia jälkeenpäin.

Gitiä käytettäessä on hyvä käydä läpi myös commit-käytännöt. Koska sanan suomenkieliset termit, kuten sisäänvienti, eivät ole erityisen vakintuneet käytössä, käytetään tässä yhteydessä jatkossa termin englanninkielistä versiota. Gitissä commit tallentaa tehdyt muutokset käyttäjän lisäämällä selitetekstillä versiohistoriaan. Jälkeenpäin on mahdollista tarkastella yksittäistä committia ja mitä rivejä muutoksessa on tullut lisää, mitä poistettu ja mitä muutettu. Versiohistoriassa kaikki commitit näkyvät listana. Committeja usein tehdään liian harvoin, joko kun niitä ei muisteta tehdä säännöllisesti, tai kun ei haluta täyttää versiohistoriaa liikaa merkinnöillä. Sen sijaan committeja pitäisi tehdä mahdollisimman usein, jotta muutokset ovat helpommin luettavia. Gitissä versionhallintahaarasta voi poimia (cherry pick) ja sulauttaa toiseen haaraan myös yksittäisen commitin, jolloin on myös tärkeää, että sulauttamisen yhdessä ei tule ylimääräisiä muutoksia. On myös hyvä käytäntö muistaa ryhmitellä commitin muutokset niiden merkityksen mukaan ja esimerkiksi tallentaa koodin siistimistä varten tehdyt muutokset omana committinaan. (Santacroce 2015, s. 100-102.)

4.5.1 Strategiat

Aiemmin valittua versionhallinta strategiaa tarkasteltiin ohjelmistosuunnitteluosaston kesken. Oli havaittu, että valinta aloittaa kehityshaarat päähaarasta alkoi helposti tuottamaan ongelmia siinä vaiheessa, kun julkaisuväliä jouduttiin pitkittämään ja siinä välissä tehtyjen muutosten määrä kasvoi. Tämä aiheutti heti huomattavasti konflikteja. Havaittiin, että työskentely on tosiaan sujuvampaa, kun uudet ominaisuushaarat otetaan aina kehityshaarasta, joten strategiasta tehtiin vielä pieni muunnos (Kuvio 5 – Lopullinen strategia versionhallintahaarojen käyttämiseksi.



Kuvio 5 – Lopullinen strategia versionhallintahaarojen käyttämiseksi

Tämän lisäksi oli huomattu, että tietyissä projekteissa tuli usein tilanteita, joissa useampi kehittäjä työskenteli omassa haarassaan saman tiedoston parissa, esimerkiksi resurssitiedostojen ja tietokantarajapintojen tapauksessa. Huomattiin, että tällaisissa tilanteissa konfliktien välttämiseksi on sujuvaa, kun kehittäjät ajoittain sulauttavat toistensa keskeneräisiä ominaisuushaaroja omiinsa joko kehityshaaran kautta, tai suoraan. Git kannustaakin käyttäjiä käyttämään versionhallintahaaroja aktiivisesti (Chacon & Straub 2014). Tämä tosin vaatii myös hyvää kommunikaatiota kehitystiimin kesken, jotta pysytään perässä siitä, kuka muokkaa mitä ja milloinkin.

4.5.2 Versionumerointi

Havaittiin, että versionnumerointi on hyvä miettiä etukäteen ja sen on hyvä olla jokseenkin yhdenmukainen projektien välillä. Versionhallinnassa on olennaista erottaa versiot toisistaan. Siinä missä tähän riittää periaatteessa jo inkrementoiva, yksinkertainen numerointi, helpottaa se silti sekä uuden version julkaisua, että ohjelmistojen riippuvuuksien linkittämistä toisiinsa, kun tiedetään jo numeroinnin perusteella, ovatko versiot keskenään yhteensopivia. (Ahlfors 2021)

4.5.3 Henkilökunnan koulutus

Henkilökunnan koulutuksessa on olennaista luoda kirjalliset ohjeet kaikille työvaiheille, kuten repositorioiden luonnille, hakemiselle ja päivittämiselle, sekä haarojen luomiselle ja yhdistämiselle. Näin voidaan varmistaa yhtenäiset työtavat, kun kaikilla on samat ohjeet. Tämä auttaa projektien yhtenäiseen rakenteeseen, sekä tietoturva-asetusten muistamiseen. Myös ongelmatilanteet on näin helpompi jäljittää ja korjata, kun voidaan kerrata jo suoritetut työvaiheet ja löytää apu nopeasti ilman internet-hakukoneita.

Tutkimuksen käyttöönottovaiheessa huomattiin, että kattavista kirjallisista ohjeista huolimatta on hyödyllistä, jos henkilökunnalla on joku kokenut käyttäjä, jonka puoleen kääntyä ongelmatilanteissa, sillä Gitin virhesanomien alkuperä voi olla välillä työlästä päätellä sellaiselle, jolle ne tulevat ensimmäisiä kertoja vastaan, mutta ratkeavat yleensä nopeasti kokemuksen karttuessa (Ahlfors 2021.)

Vaikka Gitin toiminnot, kuten kloonaus ja kommitointi löytyvät monista teksti- ja koodieditoreista sisäänrakennettuna, opeteltiin henkilöstön kanssa ensin komentojen ajaminen komentoriviltä. Vaikka toimintojen ajaminen onkin usein yksinkertaisempaa ja nopeampaa tehdä samalla työkalulla kuin ohjelmointikin, voi komentoriviä kuitenkin käyttää projektityypistä ja siihen käytettävistä työkaluista huolimatta.

4.6 Projektien siirto versionhallintaan

Ensimmäisiä projekteja siirtäessä käytiin läpi huolellisesti, mitä tiedostoja ja tietoja ei haluttu siirtää versionhallintaan. Samalla kehitettiin ja hiottiin vakiokäytäntöä, miten projektiin liittyviä ns. arkaluontoisia tietoja, kuten API-avaimia jatkossa säilytettäisiin. Projektista riippuen otettiin käyttöön ympäristömuuttujia tai yhteneväisiä, versionhallinnan ulkopuolella olevia tiedostosijainteja.

Samalla kehitettiin myös puurakenne projektien tiedostoille. Esimerkiksi kaikki projektinsisäinen lähdekoodi olisi aina saman kansiorakenteen alla, jotta jatkossa sekä projektien tarkastelu, että siihen liittyvä automatisointi olisi helpompaa. Tämän mahdollistamana kehitettiin myös yrityksen sisäinen vakiomalli gitignore-tiedostolle, jotta tietyt osiot projekteissa, kuten API-avaimien lisäksi suuret binääritiedostot jäisivät pois versionhallinnasta. Tässä hyödynnettiin olemassa olevia pohjia, kuten .NET Frameworkille tarkoitettua gitignore-mallia.

Projektien dokumentaatio käytiin myös läpi. Osa dokumentaatiosta, joka oli olennaista projektien ja niiden kehityksen kannalta jätettiin projektin repositorioon. Osa dokumentaatiosta, jota esimerkiksi tarvittiin myös kehitystyön ulkopuolella, siirrettiin muualle. Esimerkiksi sovellusten dokumentointi siirrettiin projektin Wikiin, jossa sen muutokset olivat erillään projektin muutoshistoriasta, ja dokumentaatio olisi tarkasteltavissa irrallaan repositorion lähdekoodista.

Projekteja siirrettäessä analysoitiin myös sen aiheuttamaa työnmäärää projektia kohden. Tämän pohjalta arvioitiin, että kaikkia projekteja ei välttämättä kannata edes siirtää versionhallintaan. Tähän joukkoon lukeutuivat lähinnä projektit, joita päivitettiin hyvin harvoin ja vähän kerrallaan ja tällöinkin ne työllistivät vain yhden ohjelmistokehittäjän.

4.7 Käyttöönoton tarkastelu

Kun aktiivisimmat projektit oli saatu siirrettyä versionhallintaan ja henkilöstö oli käyttänyt sitä itsenäisesti jo jonkin aikaa, pidettiin henkilöstön kesken palaveri, jossa tarkasteltiin mitä hyötyjä käyttöönotetuista työkaluista oli ja mitä pitäisi muuttaa.

Palaverissa oltiin yhtä mieltä siitä, että Gitin käyttö oli aluksi vaikea oppia. Osittain tätä olisi voitu helpottaa kattavammalla kirjallisella ohjeistuksella ja koulutuksella lähtötilanteessa, mutta toisaalta todettiin, että kirjallisen varassa työskentely olisi ollut myös työlästä, sillä loppujen lopuksi Git on vain aluksi hankala oppia. Apuna tässä kuitenkin nähtiin mahdollisuus saada tukea kokeneemmalta henkilöltä tarvittaessa.

Vaikkakin Git-konfliktit nähtiinkin hankalaksi, koettiin silti, että versioiden yhdistäminen tilanteissa, joissa konflikteja ilmeni, olisi ollut paljon hankalampaa, ellei mahdotonta ilman versionhallintaa. Erityisen hankalaksi nähtiin tietyissä projekteissa käyttöliittymämuutoksia sisältävien versioiden yhdistäminen, sillä nämä sisälsivät paljon ohjelmointiympäristön automaattisesti generoimia tiedostoja. Päätettiin, että käyttöliittymämuutoksiin olisi hyvä kehittää vastaisuudelle jokin strategia.

Loppujen lopuksi todettiin, että tähän mennessä alun vaikeuksista huolimatta versionhallinnan käyttöönotto oli ainoastaan hyödyttänyt prosesseja. Kehitystiimin kesken oltiin yhtä mieltä siitä, että ilman versionhallintaa sovelluskehittäjien olisi ollut mahdotonta työskennellä samanaikaisesti samojen projektien parissa. (Ahlfors 2021.)

4.8 Ohjelmistotestaus ja jatkuva integraatio

Toimeksiantajan henkilöstön palaverissa tarkasteltiin testiautomaatiosuunnittelua. Henkilökunnalle esiteltiin Bitbucketin tarjoamat Pipelinet, sekä erästä työpöytäsovellusta koskevan testiautomaation alustava suunnitelma ja runko.

Pipeline on eräänlainen putki, joka ketjuttaa tapahtumia, esimerkiksi lähdekoodin tarkastelun, koonnin, testauksen ja julkaisun. Kyseisillä putkilla saadaan hyvin vähällä

vaivalla ja resurssilla testattua vähintään saako lähdekoodin koottua ajettavaksi ohjelmaksi. Tämä havaittiin hyödylliseksi, sillä käytännössä oli jo huomattu, että kun kehityshaaroja yhdistellään ja sulautetaan master-haaraan, on riski, että jokin konflikti tulee käsiteltyä väärin tai jokin moduuli jää päivittämättä. Tällöin on vaarana, että ohjelmaa ei saa koottua, mikä on ongelmallista etenkin, kun strategiassa on nimenomaan määritelty, että master-haarassa viimeisin kommitointi on toimiva, julkaisukelpoinen versio ohjelmasta. (Ahlfors 2021). Pipelineillä voitaisiin myös ajaa yksikkö- ja integraatiotestejä funktiotasolla pinnan alla.

Pipelinejä ei kuitenkaan otettu heti käyttöön, sillä siinä havaittiin tarve lisäselvitykselle ja pohjatyölle, ennen kuin ne olisivat yhteensopivia ohjelmakoodien kanssa. Pipelinet ajetaan Linux-konteissa ja testattavassa sovelluksessa oli Windows-kohtaisia määrityksiä, jotka estäisivät ohjelman ajamisen. Ketjuttaminen ja testaaminen konttien avulla kuitenkin nähtiin potentiaalisesti hyödyllisenä, joten sovittiin, että käyttöjärjestelmäkohtaisia määrityksiä pyrittäisiin jatkossa parhaan mukaan välttämään, vaikka kehitettävä sovellus itse olisikin käytännössä sidottu tiettyyn käyttöjärjestelmään. Tämä tarkoittaisi muun muassa tiedostopolkujen ja vastaavien muuttujien määritys esimerkiksi ympäristömuuttujilla. (Ahlfors 2021.)

Käyttöliittymän testaamiseen käytettiin Pythonilla kirjoitettua, pywinauto-kirjastoa hyödyntävää ohjelmaa. Ohjelma käynnistää kehitettävän sovelluksen ja simuloi tilannetta, jossa oikea käyttäjä käyttää sitä. Ohjelma automatisoi napin painalluksia ja kirjoittaa tekstikenttiin tekstiä, jonka jälkeen se tarkistaa, että käyttöliittymässä näkyy oikea palaute. Mikäli jokin toiminto ei käyttäydy oletetusti, se kirjoittaa siitä lokia.

Edellä mainittu ohjelma hyödyntää Musta laatikko -metodia. Siinä annetaan sovellukselle syötteitä ja tarkistetaan mitä se tekee, tutkimatta mitä ohjelman sisällä tapahtuu. Musta laatikko - testausta voidaan käyttää kaikissa testauksen työvaiheissa, joissa ohjelmasta on käynnistyvä versio. Käytännössä tämä kattaa kaikki työvaiheet toteutuksen aloittamisesta eteenpäin yksikkö-, integrointi, järjestelmä-, ja hyväksymistestauksen tasolla. Musta laatikko -testauksella testattavat testitapaukset usein automatisoidaan toistuvan työn minimoimiseksi. (Kasurinen 2013.)

Palaverissa todettiin myös, että vaikka ohjelmiston koodiin tulisikin lisäyksiä, korjauksia tai muutoksia, ei käyttöliittymän puolelta toiminnot muutu kuitenkaan juurikaan usein, joten käyttöliittymätestejä ei tarvitsisi uusia kovinkaan usein. Koska kyseinen sovellus antaa myös palautetta kaikista käyttäjän tekemistä toiminnoista, on helppoa ja luontevaa tarkistaa musta laatikko – testeillä, että ohjelma toimii, niin kuin käyttäjä sen olettaa toimivan. (Ahlfors 2021.)

Koska ketjutukset ajetaan konteissa, joissa ei ole käytössä näyttöä, on käyttöliittymätestien ajaminen mahdotonta niissä. Tällaisiin testeihin voidaan kuitenkin käyttää muita jatkuvan integraation työkaluja. Palaverissa esiteltiin yhtenä vaihtoehtona yksi tällainen työkalu, Jenkins. Jenkins on ohjelma, jonka voi määrittää tietystä käynnistimestä hakemaan lähdekoodin repositoriosta, kokoamaan sen suoritettavaksi ohjelmaksi ja suorittamaan määrättyjä komentoja sen jälkeen. Käynnistimenä toimii esimerkiksi kommitointi tiettyyn repositorion haaraan. Suoritettavat komennot voivat esimerkiksi olla testiautomaatioajoja eri asetuksilla.

4.9 Työkalujen integrointi

Suuri osa toimeksiantajan ohjelmistoprojekteista oli NET-framework - projekteja. Microsoftin tarjoama Visual Studio mahdollistaa versionhallinnan integroimisen ohjelmointiympäristöön. Näin lähdekoodin editoiminen, muutosten kommitointi sekä versionhallintakonfliktien tarkastelu onnistuu samalla työkalulla.

Projektien dokumentointia siirrettiin paljon Bitbucketiin repositorioiden wikiin. Repositorioiden Wikit ovat omia repositorioitaan, löytyvät koodirepositorioon liitettynä verkkokäyttöliittymästä ja ovat samalla tavalla kopioitavissa käyttäjän tietokoneelle. Wiki-repositoriot tukevat merkintäkieliä, kuten Markdownia, joka on kevyt ja tarkoitettu helppolukuiseksi (Gruber n.d.).

Joidenkin projektien testauksenhallintaa yhdistettiin osittain versionhallintaan Atlassianin Jiran ja Confluencen kautta. Jiraan luotiin uusi Issue-, eli tehtävätyyppi, jota käytettiin pieniin testitapauksiin. Testitapausten kuvaus kirjoitettiin tehtävän kuvauk-

seen ja testitapaus liitettiin tarvittaessa ominaisuuksia kuvaaviin tehtäviin. Kun tehtävää oltiin siirtämässä laadunvarmistuksesta valmiiden osioon, sille suoritettiin siihen liitetyt testitapaukset, joiden tulokset kirjoitettiin tehtävän lokiin (worklog). Yksityiskohtaisemmat ja pidemmät testitapaukset kirjoitettiin Confluenceen, jonka sivuja saa linkitettyä myös Jiran tehtäviin.

Versionhallinnan mukana integroidun Jiran koettiin hyödyttäneen myös ohjelmiston elinkaarenhallintaa. Vianhallinnan todettiin kehittyneen sen myötä, sillä pienienkin vikojen ja poikkeamien kirjaus oli tehokkaampaa ja nopeampaa sen avulla. Tehtävien työstäminen ja dokumentointi koettiin myös tehokkaammaksi, kun tehtävien kuvaus ja työvaiheiden kirjaaminen tehostui ja tehtyjen töiden luettelo tallentui automaattisesti versioittain. Vaatimusmäärittelyä saatiin myös linkitettyä kehitystyöhön tehokkaammin jo yksinkertaisesti sillä, että toiminnallisia ja ei-toiminnallisia vaatimuksia joko lisättiin Jiraan Epic- ja Story- tyyppisinä tehtävinä tai linkitettiin kehitettävien ominaisuuksien tehtäväkorttiin (Issue). Näin kehitettävien ominaisuuksien vaatimuksienhallinta, testauksienhallinta ja dokumentaatio saatiin yhdistettyä niihin liittyvään kehityshaaraan.

Jiran tehtävänhallinnan kautta saatiin myös luotua uusia versionhallintahaaroja, jotka linkittyivät automaattisesti niihin liittyviin ominaisuuksiin tai bugeihin. Jiran kautta luotuna haarat saivat myös nimeensä automaattisesti etuliitteen sen mukaan, olivatko ne ominaisuus- vai korjaushaaroja. Kun taas Bitbucketissa haara sulautettiin takaisin kehityshaaraan, oli mahdollista automaattisesti merkitä haaraan liittyvä tehtävä tiettyyn tilaan, kuten valmiiksi tai testattavaksi. Jiran ja Bitbucketin työnkulun vaiheita oli mahdollista lisätä ja muuttaa tarpeiden mukaiseksi.

Työkaluja integroitaessa käytiin myös läpi erilaisia linter-työkaluja. Linter on koodinanalysointi työkalu, joka tarkastaa koodin esimerkiksi tyylivirheiden tai mahdollisten bugien varalta. Versionhallinnan pipeline-ketjutuksiin on mahdollista lisätä vaihe, joka tarkistaa lähdekoodin tällaisten poikkeamien varalta ja ilmoittaa, mikäli niitä löytyy. .NET Frameworkille sopivaa työkalua ei löydetty, sillä Visual Studiolle löytyy tällaisia työkaluja, jotka tarkastelevat koodia reaaliaikaisesti jo ohjelmointivaiheessa. Työkalun hyöty kuitenkin tiedostettiin tulevien projektien varalta.

4.10 Kommunikointi

Tutkimuksen aikana huomattiin, että vaikka versionhallintastrategia olisi mietitty miten hyvin tahansa ja henkilöstö osaisi käyttää sitä taitavasti, on kommunikaation merkitys silti hyvin tärkeä versionhallinnan tehokkaan käyttämisen edellytyksenä. Paras tapa välttää konflikteja on käytäntöjen lisäksi kommunikaatio. Tämä mahdollistaa esimerkiksi tehokkaan versionhallintahaarojen sulauttamisen, kun kehittäjät voivat poimia omaan kehityshaaraansa muiden tekemiä muutoksia.

4.11 Versionhallinnan käyttöönoton vaikutukset

Versionhallinnan käyttöönotolla havaittiin merkittäviä positiivisia vaikutuksia. Monien projektien parissa työskenteleminen ryhmässä katsottiin täysin mahdottomaksi ilman versionhallintaa. Myös muutoshistorian ylläpitämisestä saatiin sekä vaivattomampaa, että varmempaa. Aiemmin ylläpidetyt muutoslokit tulivat automaattisemmin kommitointihistorian kautta, sekä niissä tapahtuvat inhimilliset virheet esimerkiksi aikaleimojen ja päällekkäisyyksien kanssa käytännössä poistuivat. Muutosten lukemisesta tuli myös paljon nopeampaa ja helppolukuisempaa verrattuna aikaisempaan, koska Git pystyy näyttämään yksittäisen tiedoston muutoshistorian, eikä manuaalisia kommenttirivejä tarvinnut selata eikä toisaalta säilyttää yhtä pitkältä ajalta.

Virheiden jäljittämisestä tuli paljon helpompaa versionhallinnan avulla. Sen ansiosta pystyttiin esimerkiksi nopeammin ja tarkemmin jäljittämään, missä kohtaa ohjelma oli lakannut toimimasta. Versionhallinnan avulla pystyttin listaamaan mitä muutoksia sitä ennen oli tehty, sekä palaamaan takaisin siihen tilaan, missä ohjelma viimeksi toimi.

Ohjelmistojen eri versioiden ylläpitäminen oli vaivattomampaa, kun ohjelmiston useat versiot olivat kaikki samassa repositoriossa, eikä niitä tarvinnut säilyttää erikseen useamman eri kehittäjän tietokoneella. Tämän ansiosta muun muassa projektien keskinäinen linkittäminen ja päivittäminen tuli myös sujuvammaksi, kun Gitin alimoduuleita opittiin käyttämään tehokkaasti.

Versionhallinnan vaikutus prosesseihin, työnkulkuun ja ominaisuuksien läpimenoaikoihin oli myös hyvin positiivinen, sillä Jiran kautta tehtävien jakaminen, seuraaminen sekä siirtäminen kehitykseen nopeutti toimintaa huomattavasti. Tehtävien siirtäminen kehittäjältä toiselle ja jakaminen tehtävien kesken helpottui, kun kehityshaarojen kuvaus, niihin liittyvät vaatimukset ja toimintaloki ylläpidettiin Jiran kautta. Myös testaaminen erityisesti julkaisuvaiheessa helpottui Jiran Kanban-taulun avulla, kun julkaistavat ominaisuudet listautuivat kehityksen yhteydessä ja Jiran mukana saatiin myös välitettyä tietoa lokin, sekä linkitettyjen testien avulla.

5 Pohdinta

Tutkimuksen päätavoitteena oli ottaa versionhallinta käyttöön toimeksiantajalla yrityksen liiketoiminnan eri näkökulmat, erityisesti prosessitehokkuus ja tietoturvaan kohdistuvat vaatimukset huomioon ottaen. Tavoitteena oli myös tutkia versionhallinnan kautta työnkulkuun integroitavia muita työkaluja.

Tutkimuksen tuloksena alkuperäinen tavoite saatiin ratkaistua, eli toimeksiantajalle saatiin otettua käyttöön toimiva versionhallintajärjestelmä. Käyttöönotto saatiin toteutettua iteratiivisesti erilaisia toimivia ratkaisuja kokeillen ilman, että se kuormitti toimeksiantajan prosesseja. Tutkimuksen ansiosta tietoturva-asiat oli myös otettu huomioon ja dokumentoitu hyvin ennen käyttöönottoa, jolloin niihin liittyvät riskit osattiin huomioida ja ehkäistä käyttöönottovaiheessa. Menetelmänä käytettiin tutkimuksellista kehittämistyötä. Menetelmään liittyy paljon konkreettista kehittämistoimintaa, sekä tutkimuksellisten menetelmien soveltamista ja saadun aineiston analysoimista, mikä soveltui tutkimukseen hyvin.

Versionhallintajärjestelmän valitsemisessa ja käyttöönotossa onnistuttiin hyvin, sillä sen lisäksi, että itse toteutus onnistui sulavasti, huomattiin siitä myös olleen merkittävästi hyötyä yrityksen prosesseille. Tutkimuksen ansiosta saatiin syvempi ymmärrys paitsi versionhallinnan tietoturvanäkökulmista, myös erilaisista strategioista, joilla on merkittävä vaikutus tiimityöskentelyyn projektin parissa. Jälkeenpäin voi myös huomata, että käyttöönotto oli paljon tehokkaampaa, kun se oli suunniteltu etukäteen.

Tehokkuutta lisäsi myös se, että yksi henkilöstöstä koordinoi ja valvoi käyttöönoton ja se tehtiin valmiiksi luotujen standardien mukaan, jolloin työnkulku oli tehokkaampaa ja poikkeamat oli helpompi huomata ja korjata ajoissa. Erityisen tehokkaaksi huomattiin se, että ongelmatilanteessa käyttäjä pystyi saada henkilökohtaista tukea kasvokkain. Mahdollisuus henkilökohtaiseen ohjaukseen tosin on monesti pienemmän yksikön etu ja vaikeampi toteuttaa suuressa organisaatiossa.

Muut tutkittavat osa-alueet koettiin myös onnistuneiksi sikäli, että kaikilla osa-alueilla saatiin positiivisia tuloksia. ALM-integraatiolla saatiin tehostettua mm. vaatimusten-, vian- ja testienhallintaa ja sen ansiosta tehostettua työnkulkua ylipäänsä. Kehitystyön ansiosta versionhallintaa saatiin myös hyödynnettyä laadunhallinnassa mm. katselmusten osalta. Vaikka työkaluilla saatiin jollain tasolla yhdistettyä vaatimukset, ominaisuudet ja testit implementaatioon versionhallintahaarassa, sekä seuraamaan niiden etenemistä Kanban-taululla, jäi silti myös parannettavaa. Tutkitulla menetelmällä ohjelmistotestien organisointi ja raportointi oli mahdollista, mutta ei ideaalilla tasolla. ALM-käytänteistä taas esimerkiksi jäljitettävyysmatriisin luominen ei onnistunut järkevästi ilman lisätutkimusta.

Jatkuvan julkaisun, integraation ja toimituksen osalta konkreettinen hyöty jäi tavoiteltua pienemmäksi. Tämä johtui käytännössä toimeksiantajan tavallisimmista projektityypeistä ja niihin liittyvistä alustavaatimuksista sekä julkaisutavoista. Vaikka aiheesta löydettiin paljon tietoa, ei se useimmiten soveltunut silti käyttöönotettavaksi toimeksiantajalla. Suurin osa aihepiirin käytänteistä oli hyvin käyttökelpoista suuremmissa yrityksissä, tai erilaisissa projektityypeissä, kuten verkkosovelluksissa. Monet työkaluista, kuten pipeline-ketjut ja versionhallintaan integroidut automaattiset testit toivat kuitenkin verrattain vähän tehokkuutta ja lisäarvoa prosesseihin ja työnkulkuun monissa toimeksiantajan projekteissa. Jatkuvasta julkaisusta löydettiin myös paljon tietoa, jota voitaisiin mahdollisesti myöhemmin ottaa käyttöön, mutta ei saatu sovellettua toimeksiantajan nykyisiin projekteihin, joissa julkaisemisen on oltava manuaalisempaa. Vaikka jatkuvaa julkaisua, integraatiota ja toimitusta ei tutkimuksen aikana juurikaan otettu käyttöön, sen hyötyä ja käyttöönottoa saatiin tutkittua tarpeeksi kattavasti, jotta voitiin yksimielisesti hyväksyä sen käyttökelpoisuus tulevissa projekteissa.

Tutkimuksen syvyyttä rajoitti toimeksiantajan henkilöstömäärän aiheuttamat rajoitteet tutkimuksen mittavuuden järkevyyden kannalta. Alustaratkaisut, lisenssimaksut ja niiden tuomat ominaisuudet voivat vaihdella paljonkin organisaation koon mukaan, mutta niiden tutkiminen perinpohjaiselta kannalta ei ollut ohjelmistotiimin koon kannalta järkevää. Tämä ei niinkään vaikuta negatiivisesti tutkimuksen tulosten käyttökelpoisuuteen, vaan lähinnä mahdollisesti vaadittavan lisäperehtymisen määrään suuremman organisaation tapauksessa. Toisaalta huomattavasti kattavampaa tutkimustulosta olisi voinut saada, mikäli olisi voitu tutkia useampaa eri versionhallinta-alustaa samanaikaisesti rinnakkain, jotta olisi voitu verrata niiden työnkulkua ja käytettävyyttä käytännössä. Luonnollisesti kattavampaa tutkimusta versionhallinnan vaikutusmahdollisuuksista työnkulkuun olisi saatu myös, mikäli olisi tämän lisäksi voitu suorittaa vertailua käytännössä palveluntarjoajien parhailla palvelutasoilla. Tällainen tutkimus ei kuitenkaan olisi ollut toimeksiantajan kannalta järkevää, sillä usean versionhallintajärjestelmän vertaileminen samanaikaisesti työnteon ohella vaikuttaisi väistämättä negatiivisesti prosessitehokkuuteen ja parhaiden palvelutasojen lisenssihinnat taas tulisivat kohtuuttoman kalliiksi.

Tuloksia voidaan hyödyntää toimeksiantajalla muun muassa hyödyntämällä tutkimuksen aikana kehitettyä prosessia päivittäisen työskentelyn lisäksi tulevien projektien perustamisessa, sekä toisaalta yrityksen kasvaessa mahdollisesti versionhallinta-alustan vaihtamisessa, esimerkiksi toiselle palveluntarjoajalle tai itse ylläpidetylle palvelimelle. Tämän lisäksi jo tehtyä pohjatutkimusta voidaan käyttää hyödyksi erilaisten integroitavien työkalujen etsimisessä ja kehittämisessä. Tulokset ovat hyödynnettävissä myös ohjelmistoalalla yleisesti, kunhan muistetaan soveltaa tuloksia oman organisaation kriteerien mukaan, niin kuin tutkimuksen teoriaosuudessakin on painotettu, esimerkiksi alustan ja strategioiden osalta. Tutkimuksen pohdinta siitä, mitä asioita pitää ottaa huomioon pätee kuitenkin yleisellä tasolla ja eri integraatiomahdollisuuksien pohtiminen antaa pohjaa laajemmalle tutkimiselle.

Jatkokehittämistä jäi erityisesti versionhallinnan integraation osalta. Versionhallintaan on kytkettävissä paljon erilaisia työkaluja ja toimintoja, jotka parantavat työnkulkua ja prosessitehokkuutta merkittävästi. Myös jatkuvaa integraatiota, julkaisua ja toimitusta voitaisiin tutkia enemmän, sillä nykyään yleishyödyllisyyden näkökulmasta

sen merkitys on suuri ja monet versionhallinnan palveluntarjoajat tarjoavat tähän sekä sisäänrakennettuja työkaluja, että rajapintoja ulkopuolisille sovelluksille.

Tutkimuksen ajankäytön jakautumiseen vaikutti jälkeenpäin tarkasteltuna konkreettinen tarve saada versionhallinta käyttöön, sillä ennen tutkimusta oli jo käytännössä havaittu tehokkaan työskentelyn mahdottomuus ilman sitä. Koska huolimattomuus tietoturva-asioissa riskeeraisi ohjelmisto-organisaation koko liiketoimen, ei siitä voinut tinkiä. Käyttöönoton liian hätäinen suunnittelu ja toteutus vaikuttaisi prosessitehokkuuteen ja aiheuttaisi paljon työtä jälkeenpäin, koska projektien rakenne jouduttaisiin todennäköisesti alustamaan käytännössä uudelleen. Tästä syystä tutkimuksen painoa kevennettiin jälkeenpäin tarkasteltuna tarkoitettua enemmän tutkimuksen keskittymisestä erilaisiin integraatiomahdollisuuksiin ja niiden vaikutukseen alustaan ja palveluntarjoajaan. Tämä ei toisaalta tarkoita, että sen suhteen olisi tehty vääriä valintoja tai saatu epäkelpoja tutkimustuloksia, vaan että tarvetta jäi aiottua enemmän tutkimuksen ulkopuoliselle lisätutkimukselle.

Tutkimuksen pääpaino oli jakautunut sikäli oikein, että suurimmat riskit versionhallinnan käyttöönottoon tarkastelluista osa-alueista sisältyivät juuri tietoturvaan ja käyttöönottoon sekä rakenteen, että prosessien ja työnkulun strategioiden näkökulmasta. Niihin sisältyisi huonosti suunniteltuna eniten suoraa haittaa organisaatiolle, sekä eniten työtä, mikäli niitä pitäisi jälkeenpäin muuttaa. Muiden ohjelmistotuotannon käytänteiden integrointi onnistuisi jälkeenpäin inkrementoidusti helpoiten, suurimmaksi osaksi jopa alustasta riippumatta.

Lähteet

Ahlfors, J. 2020. Sisäinen palaveri versionhallinnan tavoitteista ja resursseista 5.10.2020. Ontec Oy.

Ahlfors, J. 2021. Ohjelmistokehittäjätiimin palaveri 22.1.2021. Ontec Oy.

All GitLab Features. N.d. GitLab. Viitattu 27.2.2021. https://about.gitlab.com/features/

Chacon, S. & Straub B. 2014. Pro Git. 2 p. Apress.

Cimpanu, C. 2020. Mercedes-Benz onboard logic unit (OLU) source code leaks online. Verkkoartikkeli. ZDNet. Viitattu 27.2.2021. https://www.zdnet.com/article/mercedes-benz-onboard-logic-unit-olu-source-code-leaks-online/

Driessen, V. 2010. A successful Git branching model. Verkkojulkaisu. Viitattu 13.2.2021. https://nvie.com/posts/a-successful-git-branching-model/

Ellingwood, J. 2014. Understanding the SSH Encryption and Connection Process. Verkkotutoriaali. Digital Ocean. Viitattu 28.2.2021. https://www.digitalocean.com/community/tutorials/understanding-the-ssh-encryption-and-connection-process

Friedman, N. 2020. GitHub is now free for teams. Blogijulkaisu. GitHub. Viitattu 25.3.2021.

GitHub Flow. N.d. Verkkojulkaisu. GitHub. Viitattu 18.2.2021. https://githubflow.github.io

Gruber, J. N.d. Markdown: Syntax. Verkkodokumentaatio. Daring Fireball. Viitattu 27.2.2021. https://daringfireball.net/projects/markdown/syntax#philosophy

Günther, T. 2020. On-Premise Source Code Management - 7 Git Hosting Platforms Compared. Blogijulkaisu. Tower. Viitattu 28.2.2021. https://www.git-tower.com/blog/on-premise-git-code-hosting/

Jackson, M. 2020. Exposing secrets on GitHub: What to do after leaking credentials and API keys. Blogijulkaisu. GitGuardian. https://blog.gitguardian.com/leaking-secrets-on-github-what-to-do/. Viitattu 24.2.2021

Katz, D. 2019. Why We Fail at Keeping Git Secrets. Blogijulkaisu. Viitattu 27.2.2021. https://dylankatz.com/Why-We-Fail-At-Keeping-Git-Secrets/

Li, Brian. 2021. Bitbucket vs GitHub: Which Code Repository Is Better for Your Development Projects? Blogijulkaisu. Viitattu 25.3.2021. https://kinsta.com/blog/bitbucket-vs-github/

Mørken, F. 2017. Why you should stop using Git rebase. Verkkoartikkeli. Medium. Viitattu 27.2.2021. https://medium.com/@fredrikmorken/why-you-should-stop-using-git-rebase-5552bee4fed1

Neath, K. 2010. Introducing Organizations. Blogijulkaisu. GitHub. Viitattu 25.3.2021. https://github.blog/2010-06-29-introducing-organizations/

Newdigage, A. 2018. We’re moving from Azure to Google Cloud Platform. GitLab. Viitattu 27.2.2021. https://about.gitlab.com/blog/2018/06/25/moving-to-gcp/

Nissan investigated source code exposure, says it plugged leak. 2021. Verkkoartikkeli. CyberScoop. Viitattu 27.2.2021. https://www.cyberscoop.com/nissan-source-code-leak-exposure-investigation/

Microsoft to acquire GitHub for $7.5 billion. 2018. Verkkoartikkeli. Microsoft. Viitattu 25.3.2021. https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/.

Pittet, S. N.d. Continuous integration vs. continuous delivery vs. continuous deployment. Verkkoartikkeli. Atlassian. Viitattu 2.7.2021. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

Plans and pricing. N.d. Atlassian. Viitattu 27.2.2021. https://www.atlassian.com/software/bitbucket/pricing?tab=cloud-tab

Reviewing a pull request. N.d. Verkkodokumentaatio. Atlassian. Viitattu 27.2.2021. https://confluence.atlassian.com/bitbucketserver/review-and-discuss-a-pull-request-808488540.html

Robinson, D. 2019. Git ransom incident was due to leaked credentials, say hosters. Verkkoartikkeli. DevClass. Viitattu 27.2.2021. https://devclass.com/2019/05/15/git-ransom-leaked-credentials/.

Santacroce, F. 2015. Git Essentials. Packt Publishing Ltd.

Secrets detection for Application Security. N.d. GitGuardian. https://www.gitguardian.com/secrets-detection/secrets-detection-application-security#1

Somasundaram R. 2013. Git: Version Control for Everyone. Packt Publishing Ltd. Sovelluksen elinkaaren hallinta Microsoft Power Platformin kanssa - yleiskuvaus.

Sovelluksen elinkaaren hallinta Microsoft Power Platformin kanssa – yleiskuvaus. 2020. Verkkoartikkeli. Microsoft. Viitattu 17.1.2021. https://docs.microsoft.com/fi-fi/power-platform/alm/overview-alm

Upadhyay, A. 2019. Centralized vs Distributed Version Control: Which One Should We Choose? Verkkoartikkeli. GeeksforGeeks. Viitattu 24.2.2021. https://www.geeksforgeeks.org/centralized-vs-distributed-version-control-which-one-should-we-choose/

Verona, J. 2018. Practical DevOps – Second Edition. Packt.

What features and services do I get with Azure DevOps? 2020. Verkko-opas. Microsoft. Viitattu 27.2.2021. https://docs.microsoft.com/en-us/azure/devops/user-guide/services?view=azure-devops

What is an Atlassian organization? N.d. Verkkodokumentaatio. Atlassian. Viitattu 27.2.2021. https://support.atlassian.com/organization-administration/docs/what-is-an-atlassian-organization/

Wilkes, A. 2017. Leveling up your Cloud security with 2FA and IP Whitelisting. Blogijulkaisu. Atlassian. Viitattu 27.2.2021. https://bitbucket.org/blog/big-strides-cloud-security-ip-whitelisting-required-2-step-verification-bitbucket