TURUN YLIOPISTO
Kauppakorkeakoulu

DevOps-toimintamallin yhteensovittaminen ohjelmistoarkkitehtuuriin – onnistumiset ja ratkaisut

Tietojärjestelmätieteen
pro gradu -tutkielma

Laatija:
Laura Pihamaa

Ohjaaja:
Jouni Similä, Ph.D

20.01.2023
Turku

Pro gradu -tutkielma

Oppiaine: Tietojärjestelmätiede
Tekijä: Laura Pihamaa
Otsikko: DevOps-toimintamallin yhteensovittaminen ohjelmistoarkkitehtuuriin – onnistumiset ja ratkaisut
Ohjaaja: Ph.D Jouni Similä
Sivumäärä: 56 sivua + liitteet 12 sivua
Päivämäärä: 20.01.2022

Ohjelmistotuotannossa käytetyn devOps-toimintamallin tavoitteena on automatisoida tuotannon eri vaiheita sekä nopeuttaa ohjelmiston tuotantosyklejä. DevOps-toimintamallin tarkoituksena on vastata paremmin nykyisiin liiketoiminnan tarpeisiin, jossa ohjelmistokehityksen tulisi toimia ad-hoc tyylisesti asiakaspalautteen pohjalta.

Jotta devOps voidaan ottaa onnistuneesti käyttöön ohjelmistossa, tulee ohjelmiston rakenteen sopia devOps-toimintamallin mukaiseen kehittämiseen. Tutkielman tarkoituksena on selvittää, millaiset piirteet ohjelmistorakenteessa ovat johtaneet devOpsin onnistuneeseen käyttöönottoon ja millaisia ratkaisuja devOps-implementoinnissa tulisi ottaa huomioon, jotta mallista tulee toimiva.

Tutkielman teoria esittelee devOps-toimintamallin historiaa, mallin tyypillisiä ominaisuuksia sekä devOps-tyyppisessä kehittämisessä käytettyjä työkaluja. Toisessa teorialuvussa esitellään kolme ohjelmistoarkkitehtuuria, monolittiinen, SOA- sekä mikropalvelumalli, sekä selvitetään devOpsin ja ohjelmistoarkkitehtuurin käsittelyä aiemmassa kirjallisuudessa. Tutkielma on toteutettu kvalitatiivisena tutkimuksena, ja tutkimuksen empiria on kerätty puolistrukturoiduilla haastatteluilla. Tulokset on johdettu empirian analysoinnin sekä kirjallisuudesta kerätyn aineiston pohjalta.

Avainsanat: DevOps, ohjelmistoarkkitehtuuri, CI/CD-tuotantoputki

SISÄLLYSLUETTELO

KUVIOLUETTELO

TAULUKKOLUETTELO

1 Johdanto

DevOps tulee lyhenteenä sanoista *development* (kehitys) ja *operations* (operaatiot), mikä kuvaa devOpsin tavoitetta sujuvasta yhteistyöstä IT-operaatioiden ja kehitystiimien välillä (Lwakatare ym. 2019). DevOps on osaltaan jatkumoa tietojärjestelmien kehityksessä tapahtuneeseen murrokseen, jossa ohjelmistokehitystä on alettu organisoimaan joustavammaksi (Ghantous & Gill, 2017). DevOps on laajentunut kattamaan yhteistyön lisäksi ohjelmistokehityksen integraatioita, laatua ja tuotantoa ja sen tavoitteena on parantaa organisaation asiakaspalvelua ja saavuttaa uusia liiketoimintamahdollisuuksia automatisaation avulla. (Joby, 2019).

Alun perin devOps nähtiin enemmän pienille start up -yrityksille sopivana toimintatapana, mutta myöhemmin myös isommat korporaatiot kiinnostuivat toimintatavasta sen lupaavien tuloksien myötä (Sharma 2017, 2). Harvard Business Review Analytic Servicen (2019) tekemässä tutkimuksessa 86 % tutkimukseen vastanneista oli maininnut nopean kehittämisen ja uuden ohjelmiston asettamisen tuotantoon hyvin tärkeäksi liiketoiminnan kannalta, ja tutkimukseen osallistuneista 61 % vastasi hyödyntävänsä devOps-toimintamallin periaatteita saavuttaakseen nopeampia tuotantosyklejä. Samasta tutkimuksesta kuitenkin selvisi, että vain 10 % vastanneista myönsi onnistuvansa nopeissa tuotantosykleissä.

DevOpsin käyttöönottoa varten organisaatioiden tulisikin kiinnittää huomioita uusien toimintatapojen käyttöönottoon mukauttamalla organisaatiorakennettaan ja ottamalla käyttöön uusia teknologioita ja työvälineitä. DevOpsin käyttöönotossa järjestelmän tai sovelluksen arkkitehtuurin tulisi tukea devOps-mallin mukaista kehitystä. Arkkitehtuuri on usein yksi tärkeimmistä elementeistä, kun devOpsilla halutaan saavuttaa kilpailuetua. (Babar & Shahin, 2020).

Balalaie ym. (2016) esittivät, että vaikka devOpsin implementointi onnistuu käytännössä myös monoliittisessa arkkitehtuurimallissa, on devOpsin hyödyt helpompi realisoida mikropalveluarkkitehtuurimallin avulla. Mikropalveluarkkitehtuurimallin haasteena on kuitenkin sen vaativuus, ja mallin käyttöönottaminen vaatii hyvin koulutettuja kehittäjiä sekä tuntemusta teknologioista (Babar ym., 2021). Esimerkiksi Bogner ym. (2019) esittivät kritiikkiä mikropalveluiden tutkimuksen keskittymisestä vain

muutamiin suuriin toimijoihin, kuten Amazoniin ja Netflixiin, ja siksi mikropalveluarkkitehtuuri on käytännön tasolla vielä melko tutkimaton alue.

DevOpsin käyttöönoton kannalta voidaankin pitää selkeän mallin löytämisen sijaan yhtä oleellisena tunnistaa niitä ongelmia ja haasteita, joita toimintamallin käyttöönotto aiheuttaa, ja siksi arkkitehtuuria tulisi kehittää keskittymään niihin ongelmakohtiin. (Babar ym., 2021). Tutkimusaukkona on selvittää, miten devOps-toimintamallin asettamia vaatimuksia voidaan huomioida ohjelmiston arkkitehtuurissa, kun monoliittisesta mallista lähdetään siirtymään kohti modulaarisempaa mallia.

1.1 Tutkimuskysymykset

Tutkimuksen tarkoituksena on selvittää, miten devOps-malli vaikuttaa ohjelmistoarkkitehtuuriin. Tutkimuskysymykset ovat seuraavat:

1. Millaiset ratkaisut ohjelmistoarkkitehtuurissa on koettu onnistuneiksi devOps-toimintamallin käyttöönoton kannalta ja miksi?
2. Miten ohjelmistoarkkitehtuuria voidaan suunnitella yhteensopivammaksi devOps-toimintamallin kanssa?

Ensimmäisen tutkimuskysymyksen avulla on tarkoitus selvittää, millaiset asiat ohjelmistoarkkitehtuurissa ovat tukeneet devOps-toimintamallin mukaista kehittämistä ja miksi. Toisessa tutkimuskysymyksessä selvitetään, mitä asioita suunnittelussa kannattaa huomioida, jotta arkkitehtuuri ja devOps-kehittäminen sopisi mahdollisimman hyvin yhteen. Tutkimuskysymysten tarkoituksena on keskittyä yksittäisiin ratkaisuihin arkkitehtuurissa ja devOps-kyvykkyyksissä.

2 DevOps

Ennen ketterän kehityksen (engl. agile development) menetelmien yleistymistä IT-operaatioiden ja kehittämisen välinen kommunikaatio oli melko vähäistä. Vesiputousmallissa kehittäjillä ja IT-operaatioilla oli ennalta määritellyt päivät, jolloin uutta tuotantokoodia asetettiin ensin testattavaksi ja lopulta tuotantoon. Ketterän kehityksen myötä uusia toiminnallisuuksia alettiin tuottamaan jatkuvalla syötöllä, jonka myötä myös ympäristöjen konfiguroinnit oli suoritettava nopeasti ja samalla ylläpidettävä tuotannon luotettavuutta. (Sharma 2017, 8).

DevOpsin syntyhistoria perustuu ohjelmistokehittäjä ja konsultti Patrick Deboisin kehittämiin DevOps-päiviin, jotka järjestettiin ensimmäisen kerran vuonna 2009 Belgiassa. DevOpsin alkuperäinen idea oli yhdistää ohjelmistokehitys ja prosessit, sekä lisätä yhteistyötä eri osastojen välillä (Verona 2016, 12). Yhteistyön tarkoituksena oli parantaa kommunikaatiota eri osastojen välillä, jolloin yritys voi tuottaa arvoa nopeasti ja keskeytyksettä (Ebert ym., 2016). DevOps pohjautuukin ketterään ohjelmistokehitykseen, jossa ohjelmistoa päivitetään nopeissa iteraatioissa asiakaspalautteen pohjalta (Leite ym., 2019).

Ketterät toimintamallit keskittyivät alun perin ainoastaan kehitystiimeihin, mutta hiljalleen ne laajenivat kattamaan myös IT-operaatioissa työskenteleviä tiimejä. Tästä muutoksesta muotoutui devOps. (Ghantous & Gill, 2017). Lwakatare ym. (2019) jaotteli devOpsin kolmeen kategoriaan, jossa ensimmäisessä kategoriassa devOps-käytännöt suoritetaan kehitystiimien ja IT-operaatioiden yhteistyönä ja kaksi muuta kategoriaa viittaavat käytäntöihin, jotka suorittaa vain kehitystiimi tai vain IT-operaatioiden tiimi. IT-operaatioiden ja kehitystiimin yhteistyönä suorittamat työt keskittyvät pääsääntöisesti käännösten automatisointiin käännöstuotantoputken (engl. deployment pipeline) avulla. Kääntämisen lisäksi yhteisiin työtehtäviin kuuluu ohjelmiston toiminnan monitorointi sekä tietoturva.

DevOpsilla on organisaatioissa perinteisesti pyritty lyhentämään ohjelmakoodin tuotantosyklejä sekä tehostamaan ongelmien ratkaisua (Leite ym., 2019). Ghantousin ja Gillin (2017) tekemän tutkimuksen mukaan devOpsin tärkeimpiä elementtejä ovat kommunikaatio ja yhteistyö tiimien välillä, automatisoitu tuotantoputki (engl. automated pipeline) sekä jatkuva kääntäminen (engl. continuous deployment). Heidän mukaansa

devOpsia voidaan kuvailla automaatiolähtöisenä toimintatapana ohjelmistokehitykseen. DevOps voidaan kuitenkin nähdä automaation lisäksi kulttuurisena muutoksena, jossa kehitystiimien on ymmärrettävä kehittämisen lisäksi tuotantoympäristöjen ominaisuuksista, testaamisesta sekä monitoroinnista, samalla kun IT-operaatiot ovat tietoisempia siitä, millaista ohjelmakoodia on tulossa, miten se vaikuttaa järjestelmään, miten järjestelmät versioidaan ja miten automatisoidaan järjestelmien ylläpitoa (Sharma, 2017). DevOpsin käyttöönotossa myös johtamisella on merkittävä rooli. Johtamisen tehtävänä on luoda uusia, yhteisiä arvoja ja toimintatapoja koko organisaatioon sekä eri IT-tiimeihin. Lisäksi johtamisella tulisi parantaa kommunikaatiota ja palautteenantoa sekä oman tiimin sisällä että eri tiimien välillä. (Ravichandran & Waterhouse, 2016).

Ravichandran ja Waterhouse (2016) huomauttavat, että devOpsissa järjestelmiä tulisi ajatella kokonaisuuksina. DevOps on konseptina hyvin monitahoinen, ja useimmiten onnistuneet devOps-implementaatiot ovat vaatineet muutoksia organisaatiorakenteeseen, järjestelmäarkkitehtuuriin sekä organisaation prosesseihin (Lwakatare ym., 2019). Baskaradan ym. (2018) tutkimuksen haastateltavat nostivat esille myös sen, että devOpsin implementointi IT-alan ulkopuoliseen liiketoimintamalliin voi vaatia tavallista suurempia muutoksia organisaatiorakenteessa, sillä devOpsia on perinteisesti sovellettu IT-organisaatioissa, joissa liiketoimintamalli perustuu IT-palveluihin.

2.1 DevOpsin ominaisuudet

Jossain kirjallisuudessa devOpsin sanotaan alun perin muovautuneen jatkuvan kääntämisen periaatteista, ja siksi suuri osa devOpsin toimintatavoista (kuten siiloutumisen ehkäisy, automaattiset prosessit julkaisussa ja infrastruktuurissa sekä monitorointi) tukee jatkuvan kääntämisen onnistunutta käyttöönottoa (Lwakatare ym., 2016). DevOps jaetaankin usein kahteen ominaisuuteen, jatkuvaan kääntämiseen sekä jatkuvaan integraatioon, ja voidaan esittää, että kaikki muut toiminnot näiden ympärillä ovat vain näitä kahta ominaisuutta tukevia toimintoja (Sharma 2017, 11). Jatkuvien toimintatapojen hyötyinä ovat palautteen saamisen nopeutuminen, asiakastyytyväisyyden ja kommunikaation paraneminen sekä manuaalisen työn väheneminen (Babar ym., 2017).

Jatkuvaa kääntämistä ei ole mahdollista implementoida ilman toimivaa kommunikaatiota IT-operaatioiden ja kehitystiimien välillä, ja siksi devOps-toimintamallin hyödyntämisen voi sanoa olevan välttämätöntä, jos organisaatio haluaa

hyödyntää jatkuvaa kääntämistä osana tuotantoputkea (Lwakatare ym., 2016). Aiemmin IT-operaatiot ja kehitystiimit toimivat siiloissa niin, että IT-operaatioiden tehtävänä oli huolehtia ohjelmiston muutoksista tuotantotasolla ja ohjelmistokehittäjien tehtävänä oli tuottaa uusia ominaisuuksia ohjelmistoon. Perinteinen tapa kommunikaatioon oli tikettijärjestelmä, jossa IT-operaatiot ottivat manuaalisesti kehitystiimien tuottamia tikettejä käsittelyyn. Toimintatavan etuna voidaan nähdä ohjelmiston stabiilius, mutta samalla uusien ominaisuuksien käyttöönottaminen on hidasta, eikä se sovellu ketteriin menetelmiin, jossa ohjelmistoa on tarkoitus päivittää nopeissa iteraatioissa. DevOpsin tarkoituksena on kattaa tämä siilo niin, että kääntäminen ja julkaisu voidaan automatisoida ja samalla nopeuttaa tuotantosykliä. (Leite ym., 2019).

2.1.1 Jatkuva integraatio

Jatkuva integraatio (engl. continuous integration) on käsitteenä saanut alkunsa Xtreme Programming -metodologiasta, jonka juuret ovat ketterässä kehittämisessä, joskin nykyään se on sovellettavissa laajasti eri agile-menetelmissä (Aurum ym., 2015). Jatkuvan integraation tarkoituksena on integroida työn alla olevia ominaisuuksia mahdollisimman usein osaksi suurempaa kokonaisuutta. Näin ohjelmistoyritykset voivat lyhentää julkaisuaikojaan sekä parantaa ohjelmiston ja tiimien tehokkuutta. (Babar ym. 2017). Jatkuvassa integraatiossa muutoksia ei säilötä omalla työasemalla, vaan ne jaetaan tiimin kesken ja koodin toimivuutta tarkastellaan jatkuvasti (Virmani, 2015). Jatkuvaan integraatioon kuuluu ohjelmistokoodin integroinnin lisäksi ohjelmiston paketointi (engl. build) sekä ohjelmiston testaaminen (Babar ym. 2017) ja tämän prosessin tuloksena ohjelmistokoodia voidaan ylläpitää jatkuvasti tuotantovalmiina (Aurum ym., 2015).

Jatkuvassa integraatiossa muutokset tulisi jakaa oman komponentin lisäksi koko tuotteen integraatiotasolla (Virmani, 2015). Applikaatioiden ympäristöön kuuluu pääsääntöisesti myös muita järjestelmiä, komponentteja sekä applikaatioita, joiden kanssa uuden ominaisuuden tulisi pystyä keskustelemaan ja toimimaan. Perinteisessä ohjelmistokehityksessä integraatiot suunniteltiin vasta sitten kun applikaatio oli täysin valmis, mutta tämän työtavan ongelmana olivat korkeat kustannukset sekä toiminnan epävarmuus. Ketterän kehityksen myötä integraatioita alettiin kehittämään ja testaamaan niin usein kuin siihen tuli mahdollisuus, jonka myötä mahdolliset ongelmat pystytään tunnistamaan jo aiemmassa tuotantovaiheessa. (Sharma 2017, 11–12). Kun ongelmatilanteet löydetään mahdollisimman aikaisessa vaiheessa, niiden korjaaminen on

helpompaa, sillä ongelmien etsiminen ja korjaaminen vaikeutuu aina integraatioiden aikavälin kasvaessa. Jatkuva integraatio parantaa kehitystiimin välistä läpinäkyvyyttä sekä koordinaatiota, joka taas vähentää integraatioista johtuvia ongelmia. (Yarlagadda, 2018).

Jatkuva integraatio voidaan implementoida esimerkiksi teknologialla, joka sisältää koodirepositorioon puskettavien muutosten automaattisen havainnoinnin. Kun repositoriossa havaitaan muutoksia, käännetään alkuperäinen lähdekoodi sekä muutokset, ja käännöksen onnistuessa voidaan koodi viedä yksikkötestattavaksi. Onnistuneiden yksikkötestauksien jälkeen ohjelmakoodi voidaan siirtää testiympäristöön, jossa suoritetaan hyväksymistestit (engl. acceptance test), ja onnistuneiden testien jälkeen muutokset voidaan julkaista kaikille kehitystiimien jäsenille, sekä ilmoittaa raportilla tarvittavat tiedot, kuten esimerkiksi testien lukumäärä ja paketin numero (engl. build number). (Stolberg, 2009).

2.1.2 Jatkuva kääntäminen

Jatkuva kääntäminen (engl. continuous deployment) voidaan ajatella jatkuvaa integraatiota seuraavana käytäntönä, jossa rakennettu applikaatio siirretään integraatioiden jälkeen tuotantoympäristöön, jos se läpäisee vaaditut hyväksyntätestit (Sharma 2017, 17–18; Pulkkinen, 2013). Jatkuva kääntäminen laajentaa jatkuvan integraation käytäntöjä toimittamalla ohjelmistoa tuotantoon asti onnistuneiden integraatioiden jälkeen, ja siksi ilman toimivaa jatkuvan integraation mallia ei jatkuvaa kääntämistäkään voida suorittaa (Aurum ym., 2015).

Jos jatkuvan kääntämisen prosessia verrataan tavalliseen ohjelmistokehitykseen, eroaa jatkuva kääntäminen nopeammalla iteraatiotahdilla, sillä uusia muutoksia käännetään päivittäin. Lisäksi jatkuva kääntäminen minimoi tuotannossa tapahtuvia riskejä, sillä ongelmat ovat pienempiä, kun myös muokkaukset ovat pienempiä. Jatkuvan kääntämisen avulla on helpompi tunnistaa kehitystä vaativia toiminnallisuuksia, sillä asiakaspalautteeseen voidaan reagoida nopeammin kuin perinteisessä ohjelmistokehityksessä. (Aurum ym., 2015).

Jatkuva kääntäminen ei ole niinkään suoraviivainen prosessi, vaan se kuvaa enemmänkin organisaation kyvykkyyttä ottaa uusia ominaisuuksia käyttöön missä tahansa ympäristössä aina tarpeen vaatiessa. Jatkuvissa käännöksissä julkaistava osa voi

täyden applikaation sijaan olla myös applikaation jokin komponentti, sisältö, konfiguraatio tai ympäristö, johon applikaatio julkaistaan, tai näiden eri asioiden yhdistelmä. (Sharma 2017, 17–18). Jatkuva kääntäminen eroaa jatkuvasta julkaisusta (engl. continuous delivery, CDE) siinä, että se ei pyri ainoastaan pitämään ohjelmiston tilaa aina julkaisu valmiina, vaan se pyrkii myös automatisoimaan koodin toimittamista tuotantoon, kun koodi on läpäissyt tarvittavat automaattiset testit. (Lwakatare ym., 2016).

Jatkuvassa kääntämisessä ei tulisi olla manuaalista työtä, vaan kaiken työn tulisi olla automaattista (Babar ym., 2017). Esimerkiksi pilvipohjaisissa palveluissa jatkuva julkaiseminen voidaan sisällyttää suoraan jatkuvan integraation palvelimelle. Tässä tapauksessa palvelimen virtuaalikoneelle on asennettu automaattisia laukaisijoita, jotka suorittavat käännökseen vaadittavat muutokset ympäristössä. Käännösautomaation tarkoituksena on parantaa käännöksen luotettavuutta sekä toistettavuutta. (Lwakatare ym., 2019). Automaattinen kääntäminen hyödyntää usein *infrastructure-as-a-code* -konseptia, sillä se vaatii palvelimen, yhteyden ja muiden elementtien keräämistä johonkin tiedostoon, johon sekä ohjelmoijilla että IT-operaatioilla on pääsy (Leite ym., 2019).

2.1.3 Ketterä kehittäminen

Tieteellisessä kirjallisuudessa devOps on esitelty yhtenä ketterän kehittämisen tyyppinä tai osa-alueena. DevOpsin implementoiminen osaksi perinteisempää vesiputousmallia voidaan nähdä jossain määrin mahdottomana, sillä devOpsin tavoitteena on tuottaa uusia päivityksiä nopeissa iteraatioissa. Ketterä kehittäminen ja devOps jakavatkin paljon yhteisiä ominaisuuksia, kuten kommunikaation kehittäminen joko kehittäjien ja IT-operaatioiden välillä tai kehittäjien ja asiakkaan välillä, järjestelmän kehittäminen yhtenä kokonaisuutena, nopeat iteraatiot sekä ongelmanratkaisu. (Lwakatare ym., 2016). Yksi devOpsin alkuperäisistä tarkoituksista olikin mahdollistaa ketterä kehittäminen hyödyntämällä erilaisia viitekehyksiä, kuten jatkuvaa integraatiota ja kääntämistä sekä mikropalvelumallia (Yarlagadda, 2018).

2.2 DevOps työkaluja

DevOps-toimintamallin kehittämiseen on saatavilla useita erilaisia työkaluja, jotka automatisoivat kehittämistä ja mahdollistavat ketterää kehittämistä. DevOps-työkalut voivat liittyä esimerkiksi versionhallintaan, testaamiseen, monitorointiin, tietoturvaan sekä konfiguraatioidenhallintaan. Tässä kappaleessa esittelemme versionhallintaan,

konfiguraatioiden hallintaan ja käännösten hallintaan liittyviä työkaluja, sekä käymme läpi tarkemmin CI/CD-putkea ja sen toimintaa devOps-ympäristössä.

2.2.1 Versionhallinta

Versionhallintajärjestelmällä tarkoitetaan järjestelmää, joka pitää kirjaa kaikista dokumenttiin tulleista muutoksista. Ohjelmistokehityksen yhteydessä versionhallintajärjestelmästä puhuttaessa hallinnoidaan versionhallintajärjestelmillä koodiin tai skripteihin tulleita muutoksia. (Dubaria ym., 2018). Myös operaattorit voivat hyödyntää versionhallintajärjestelmiä tallentamalla automaatioskriptejä tai infrastruktuuriin liittyviä artefakteja, jolloin versionhallintajärjestelmä edistää kommunikaatiota myös kehittäjien ja operaattoreiden välillä (Leite ym., 2019).

Versionhallintajärjestelmien avulla useampi ohjelmistokehittäjä voi työskennellä saman koodin parissa yhtäaikaisesti. Versionhallintajärjestelmät mahdollistavat eri haarojen (engl. branch) luomisen uusien toiminnallisuuksien kehittämisen ajaksi, ja haarat voidaan yhdistää päähaaraan, kun toiminnallisuus on valmis. (Dubaria ym., 2018). Versionhallintajärjestelmän tarkoituksena on parantaa kehittäjien välistä kommunikaatiota (Leite ym., 2019).

Kun haluttu toiminnallisuus yhdistetään päähaaraan, versionhallintajärjestelmä nostaa yhdistämispyynnön (engl. merge request), joka laukaisee automaattiset testit ja ne läpäistyään koodi yhdistetään päähaaraan. Näin versionhallintajärjestelmillä on merkittävä rooli jatkuvassa integraatiossa. (Gaba ym., 2019). Tunnettu versionhallintajärjestelmä on Git-versionhallintajärjestelmä, jota voidaan hyödyntää useampien alustojen, kuten GitLabin ja GitHubin kanssa (Leite ym., 2019). Muita versionhallintajärjestelmiä ovat esimerkiksi CVS ja Bitbucket.

2.2.2 Konfiguraatioiden hallinta

DevOpsin yksi toiminnallisuus on konfiguraatioiden automaattinen ajo yksinkertaisten komentojen avulla. Etuna automaattisessa ajossa on käännösten jatkuva ja ajantasainen päivittyminen jokaisen tapahtuman jälkeen, jolloin ajo yhdessä koneessa päivittää kaikki koneet. DevOps konfiguraatiotyökaluja ovat muun muassa Puppet, Salt Stack ja Ansible. (Yarlagadda, 2021). Konfiguraatiotyökalut mahdollistavat siirrettävyyttä ja idempotenssia skripteihin verrattuna, sillä paketteja voidaan purkaa vasta ajovaiheessa ja

ajolaukaisuun käytetyn skriptin sijaan palvelun oletus on olla aina ajossa (Leite ym., 2019).

Konttiteknologiassa konfiguraatioita voidaan tuottaa myös hyödyntämällä valmiina olevia kuvia (engl. images), joita on tallennettuna kuvarepositorioon. Konfiguraatioita voidaan muokata kuvan paketointi vaiheessa. Tässä tapauksessa kuville on varattava oma repositorio, johon kuvat varastoidaan. Kuvarepositoriosta voidaan noutaa kuvia ajoon sekä tuotanto- että testiympäristöön, ja jokaisella sovelluksella tulisi olla oma kuvakonttinsa. (Balalaie ym., 2018).

2.2.3 Käännösten hallinta

Yleensä käännöstenhallinnan työkalut yhdistetään tuotantoputken käännösvaiheisiin, jolloin mahdollistetaan jatkuva kääntäminen. Leite ym. (2019) esittelivät kolme vaihtoehtoa käännöstenhallinnalle. Ensimmäisessä vaihtoehdossa käännöstenhallinta voidaan suorittaa pilvipalvelussa, kuten AWS (Amazon Web Services) tai Azure, jotka tarjoavat infrastruktuuripalveluita kuten virtuaalikoneiden, tietokantojen sekä jonojen hallintaa IaaS-mallilla (infrastructure-as-a-service). Toinen vaihtoehto on hyödyntää PaaS-palveluita (platform-as-a-service), jossa PaaS-palvelun tuottajan vastuulla on tuottaa alusta, jolle sovellukset asennetaan. Kolmas vaihtoehto on hyödyntää konttiteknologiaa. Pilvipohjaisissa palveluissa käännöstenhallinta voidaan asentaa myös osaksi CI-putkea määrittelemällä laukaisijoita, jotka ajavat käännökset testi- tai tuotantoympäristöön. Automaattisen käännöstenhallinnan etuna on sen luotettavuus ja stabiilius. (Lkwatare ym., 2019).

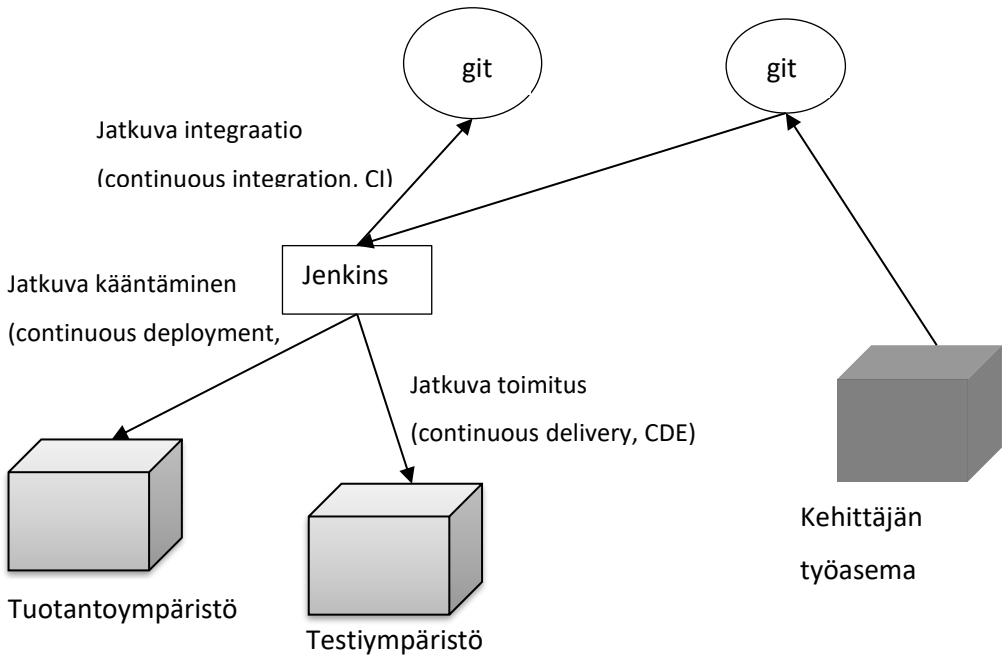
Käännöstenhallinnan mekanismit voivat vaihdella aina käännöstenhallinnan strategian mukaan (Lkwatare ym., 2019). Dark horse, vaiheittaiset roll-outit ja beta käyttäjät strategiat perustuvat siihen, että uudet ominaisuudet julkaistaan ainoastaan tietyille käyttäjille. Dark horse -strategiassa ominaisuudet julkaistaan testaajille, vaiheittaisissa roll-outeissa ainoastaan osalle käyttäjistä ja beta users -strategiassa ominaisuudet julkaistaan vapaaehtoisille käyttäjille, jotka ovat ilmoittautuneet halukkaiksi testaamaan uusia ominaisuuksia niiden saapuessa. Näiden strategioiden lisäksi ominaisuuksiin voidaan lisätä on/off-toiminnallisuus, jonka avulla ominaisuus voidaan kytkeä pois päältä tarvittaessa. Tätä strategiaa kutsutaan feature flag -strategiaksi. Blue-green-strategialla viitataan strategiaan, jossa nopea palautus (engl. roll-back) mahdollistetaan käyttämällä kahta eri ympäristöä, niin että toiseen ympäristöön

käännetään uudet ominaisuudet ja toisessa säilytetään vanhaa versiota. Myös A/B-testausta ja monitorointimetriikkaa voidaan käyttää. (Pulkkinen, 2013).

2.2.4 CI/CD-työkalut

Jatkuvaan integraatioon ja jatkuvaan kääntämiseen on kehitetty työkaluja, jotka tukevat käännöstyökalujen toimintaa. Näitä työkaluja kutsutaan nimellä CI/CD-työkalut. Tunnettuja CI/CD-työkaluja ovat muun muassa Jenkins ja GitLab. Molemmat reagoivat koodirepositorioon tuleviin muutoksiin, jonka jälkeen ne onnistuneen käännöksen jälkeen lähettävät käännöksen joko testi- tai tuotantoympäristöön. Epäonnistuessaan ympäristöön voidaan palauttaa alkuperäinen käännös. (Gaba ym., 2019).

CI/CD-tuotantoputken tarkoituksena on nopeuttaa kehitystä ja helpottaa uusien ominaisuuksien lisäämistä sekä virheiden jäljitystä ja korjaamista. CI/CD-tuotantoputkessa CI-osa on suoritettu hyväksytysti, kun koodi on lisätty, testattu ja integroitu onnistuneesti osaksi päähaaraa. CDE-osassa koodi pusketaan automaattisesti testiympäristöön, kun taas CD-osassa koodi pusketaan tuotantoympäristöön loppukäyttäjille käytettäväksi. (Dubaria ym., 2018). Alla olevassa kuvassa on havainnollistettu CI/CD-putken toimintaa (kuva 1).



Kuva 1 CI/CD-tuotantoputki (mukaillen Dubaria ym. 2018)

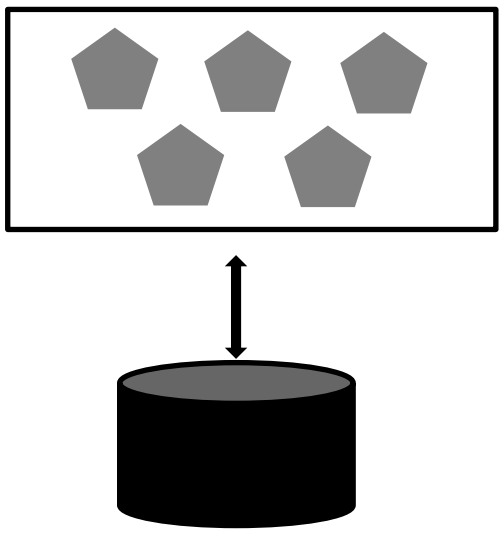
3 Ohjelmistoarkkitehtuuri

Ohjelmistoarkkitehtuurin tehtävänä on kuvailla ohjelmiston kommunikaatiota ja koordinaatiota eri elementtien välillä sekä hahmottaa tarvittavia teknisiä ja toiminnallisia tarpeita (Jaiswal, 2019). Ohjelmistoarkkitehtuuria voidaan kuvailla ohjelmiston rakenteen eri osien välisinä suhteina sekä osien ominaisuuksina (Bolscher & Daneva, 2019). Arkkitehtuuri voidaan ajatella sekä arkkitehtuurisina konsepteina (kuten malleina) että teknologisina ratkaisuina (kuten työkaluina ja ohjelmointikielinä) (Shahin ym., 2016) ja tässä tutkimuksessa käsitellään arkkitehtuuria molemmista näkökulmista.

3.1 Ohjelmistoarkkitehtuurin historiaa

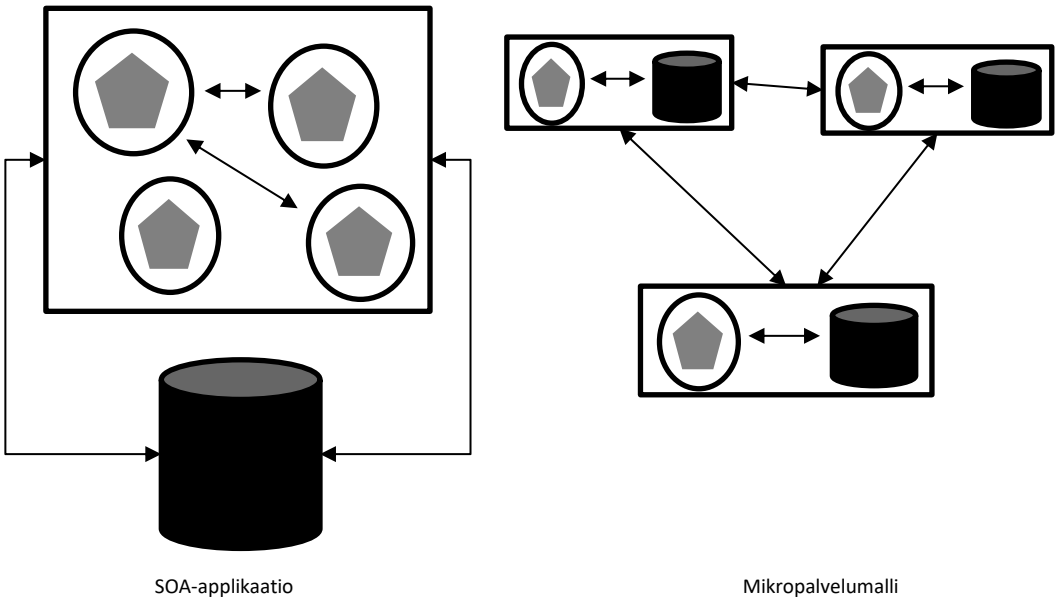
Ohjelmistoarkkitehtuurin käsitteellä tarkoitettiin vielä 1980-luvulle asti lähinnä järjestelmärakennetta, jolla kuvattiin koneiden fyysisiä ominaisuuksia. Arkkitehtuuri alkoi myöhemmin käsittämään yhä laajemmin operaatioiden ja toiminnallisuuksien välisiä suhteita, ja nykyään ohjelmistoarkkitehtuuriin on olemassa useita erilaisia suunnittelumalleja. (Kruchten ym., 2006).

Arkkitehtuurimalleista monoliittisella arkkitehtuurilla viitataan malliin, jossa jokainen toiminnallisuus on nivottu yhteen käännökseen (Kruchten ym., 2006). Monoliittiset arkkitehtuurin ongelmana on skaalautuvuus, integroituvuus sekä ylläpidon kustannukset, ja siksi ne eivät vastaa nykyisen digitaalisen liiketoiminnan tarpeisiin, jossa muutoksia tulee nopeaan tahtiin (Ravichandran & Waterhouse, 2016). Alla olevassa kuvassa (kuva 2) on havainnollistettu monoliittista arkkitehtuurimallia.



Kuva 2 Monoliittinen arkkitehtuurimalli (mukaillen Astudillo ym., 2019)

Monoliittisen arkkitehtuurimallin laajuudesta johtuvia ongelmia alettiin myöhemmin selvittämään siirtymällä palvelupohjaiseen arkkitehtuuriin (engl. service oriented architecture, SOA), jossa useammat loppukäyttäjien applikaatiot voivat kommunikoida saman palvelun kanssa. SOA:sta siirryttiin vielä myöhemmin mikropalveluarkkitehtuurimalliin, jonka tarkoituksena on pilkkoa palvelut omiksi yksiköikseen niin, että SOA:sta poiketen ne eivät jaa yhteistä tietokantaa, ja ovat yhteydessä toisiinsa löysillä liitoksilla (engl. loosely coupled). (Newman, 2015). Mikropalvelumalli kiertää monoliittisen mallin ongelmia hyödyntämällä palvelupohjaista arkkitehtuuria ja sen hyviä puolia (Bolscher & Daneva, 2019). Alla olevassa kuvassa (kuva 3) on SOA- ja mikropalvelumalli havainnollistettuna rinnakkain.



Kuva 3 SOA-malli ja mikropalvelumalli (mukaillen Astudillo ym., 2019)

SOA:n ja mikropalvelumallien eroa voidaan kuvailla niin, että kun SOA:n ideana on jakaa sovellusten välillä mahdollisimman paljon eri resursseja (esimerkiksi komponentteja, viestintää ja palveluita), pyrkii mikropalvelumallissa eri sovellukset jakamaan mahdollisimman vähän resursseja ja oletuksia keskenään (Liang ym., 2020). SOA ja mikropalveluarkkitehtuurimalli eroavat toisistaan myös rakenteeltaan, sillä mikropalveluarkkitehtuurimallissa ohjelmiston eri osat kommunikoivat keskenään suoraan käyttäen eri standardeja kuten HTTPS tai REST, kun taas SOA on rakennettu palveluväylien (engl. Enterprise Service Bus, ESB) varaan. Lisäksi SOA hyödyntää

olemassa olevia palveluita monoliittisesta arkkitehtuurista, kun taas mikropalvelumalliin monoliittiset applikaatiot harvoin sopivat. (Baskarada ym., 2018).

Mikropalvelumallia voidaan kuvailla pilvinatiivina arkkitehtuurimallina, ja jokainen ohjelmiston osa on käännettävissä eri alustalle ja eri teknologiapinolle (engl. technology stack). Jokainen palvelun osa edustaa yhtä liiketoiminnan ominaisuutta, joka voi hyödyntää useampia tietovarastoja sekä ohjelmointikieliä. Palvelun osia kehittävät yleensä pienet tiimit. (Balalaie ym., 2016).

3.2 DevOps ohjelmistoarkkitehtuurissa

Mishra & Otaiwi (2020) esittivät, että ohjelmistoarkkitehtuuri on toimivan devOpsin perusta. He nostivat esille, että ohjelmistoarkkitehtuuri ja devOps kytkeytyvät toisiinsa erityisesti jatkuvan kääntämisen tavoitteissa. Jatkuvan kääntämisen lisäksi devOps-käytänteisiin kuuluva ketterä kehittäminen ei ole mahdollista ilman toimivaa järjestelmäarkkitehtuuria.Vaikka järjestelmäarkkitehtuurin rooli on hyvin oleellinen devOpsin kannalta, se jää usein jälkeen ketterien kehitystapojen kehityksestä. Järjestelmäarkkitehtuurilta vaaditaan enemmän joustavuutta, ja siksi arkkitehtien tehtävänä on kommunikoida ohjelmistokehitystiimien kanssa mitä teknologioita käytetään ja miksi se on tärkeää kokonaiskuvan kannalta. (Ravichandran & Waterhouse., 2016).

DevOpsin ja arkkitehtuurin suhde voidaan nähdä niin, että organisaation implementoidessa uuden arkkitehtuurimallin, se ottaa samalla käyttöön devOps-toimintamallin ylläpitääkseen uutta arkkitehtuurimallia (Liang ym., 2020). Esimerkiksi mikropalveluarkkitehtuurimallien monimutkaisuus saattaa johtaa organisaation implementoimaan erilaisia devOps-käytäntöjä, sillä mikropalveluiden vaihtelevat teknologiat ja kompleksiset rakenteet eivät ole organisoitavissa muilla menetelmillä (Baskarada ym., 2018). DevOpsin avulla organisaatio voi vähentää mikropalvelumallin käyttöönotosta muodostuvia haasteita, ja devOps voi toimia mikropalvelumallin prosessiviitekehyksenä (Liang ym., 2020). Muutos voi kuitenkin tapahtua myös niin, että organisaatio päättää ensin implementoida devOps-käytäntöjä ja sen myötä myös arkkitehtuuria tulee muokata devOps-käytänteisiin sopivaksi. Tässä tutkimuksessa huomioidaan sekä arkkitehtuurimigraation yhteydessä luodut devOps-kyvykkyydet, että devOps-toimintamallin vaatimusten huomiointi arkkitehtuurissa yleisellä tasolla.

DevOpsin ja arkkitehtuurin soveltuvuudessa isoimmat haasteet ovat yleensä arkkitehtuurin skaalautuvuudessa, ja haasteita aiheuttaa erityisesti monoliittinen järjestelmä, jossa eri osat ovat tiukasti liitoksissa toisiinsa. Vaikka muuten monoliittinen järjestelmä pilkottaisiinkin modulaarisemmaksi, voi lopullinen haaste aiheutua yhteisestä keskitetystä tietokannasta, joka on monoliittiselle järjestelmälle tyypillinen. (Bolscher & Daneva, 2019). Bolscherin ja Danevan (2019) tutkimuksessa mikropalvelumalli nostettiin kaikissa artikkeleissa devOpsia tukevaksi arkkitehtuurimalliksi. Tutkimuksen artikkeleissa mikropalvelumallin kuvailtiin sisältävän useita jatkuvan kääntämisen mahdollistavia tekijöitä, kuten esimerkiksi mallin modulaarisuus. Liang ym., (2020) nostivat esille devOpsin ja mikropalvelumallin hyötyinä niiden yhteneväiset piirteet, kuten isojen kokonaisuuksien pilkkomisen pienempiin osiin, työmäärän jakamisen poikkiosaaviin tiimeihin, luotettavuuden. skaalautuvuuden, virheresilienssin sekä tiimien ohjautuvuuden. Kun esimerkiksi monoliittisessa arkkitehtuurissa pienen osan muuttaminen vaatii koko sovelluksen uudelleen kääntämisen ja julkaisun, voidaan mikropalvelumallissa keskittyä aina yhteen sovelluksen osaan kerrallaan, ja siksi mikropalvelumalli mahdollistaa jatkuvan kääntämisen mukaista kehittämistä (Astudillo ym., 2019).

Baskaradan ym. (2018) tutkimuksessa haastateltavat olivat kuitenkin melko yksimielisiä siitä, että mikropalvelumalli ei sovellu monimutkaisiin yritysten tietojärjestelmiin, kuten toiminnanohjaus- tai asiakastietojärjestelmiin, sillä tällaisten järjestelmien pitäisi olla luonteeltaan vakaita, eikä vaatia nopeita muutoksia. Kommunikaatioon tarkoitetut järjestelmät, järjestelmät, joilla organisaatiot erottuvat muista sekä innovatiiviset järjestelmät soveltuivat haastateltavien mukaan mikropalvelumalliin paremmin. Mikropalvelumallin mukaisten konttien käyttö ei myöskään itsessään ratkaisen kaikkia vanhojen järjestelmien (engl. legacy applications) ongelmia, ja esimerkiksi konttien alustariippumattomuus ei aina sovellu suoraan vanhoihin järjestelmiin (Ravichandran & Waterhouse, 2016). Myös Bolscer & Daneva (2019) nostivat tutkimuksessaan esiin riippuvuuksiin ja virheiden jäljityksiin liittyvät ongelmat, jotka voivat estää monoliittisen järjestelmän purkamisen komponentteihin.

Babar & Shahin (2020) esittivät, että pelkän mikropalvelumallin sijaan arkkitehtuurin modulaarisuus on tärkein tekijä devOpsin onnistuneessa hyödyntämisessä. DevOps-toimintamallia implementoitaessa arkkitehtuurissa tulisi heidän mukaansa keskittyä löysiin liitoksiin (engl. loosely coupled) sekä arkkitehtuurin iteroimiseen (engl.

deployability), testaukseen, tukeen ja modifiointiin. Bolscher & Daneva (2019) nostivat esiin näiden ominaisuuksien lisäksi komponenttien tilattomuuden (engl. stateless component), jolloin komponentteja voidaan pysäyttää ja käynnistää ilman ongelmia, sekä arkkitehtuurin ohjelmakoodin tuotannon automatisoinnin ja testattavuuden.

Löysiin liitoksiin perustuvassa arkkitehtuurissa palvelut toimivat itsenäisesti, sillä ne jakavat keskenään vain pienen määrän yhteisiä oletuksia. Löysästi liittoutuneet palvelut eivät ota kantaa toistensa rakenteisiin, vaan ainoastaan kommunikoivat keskenään ja siksi palveluiden rakennetta on mahdollista muuttaa ilman, että se vaatii muutoksia niihin applikaatioihin jotka sitä hyödyntävät. Löysät liitokset ovat tiukasti liittoutuneita järjestelmiä luotettavampia monimutkaisissa järjestelmissä, sillä paikallinen virhe ei laajennu koskettamaan koko järjestelmää. (Pautasso & Wilde, 2009).

Mikropalvelumalleista ja devOpsista puhuttaessa voidaan nostaa esille kaksi teknologiaa, jotka tukevat hyvin devOps-mallin mukaista kehittämistä. Nämä teknologiat ovat konttiteknologia (engl. container) ja pilvinatiivit sovellukset. Konttiteknologia on virtuaalikoneisiin verrattuna kevyt, ja siksi se on yhä suositumpi teknologia mikropalvelumallin mahdollistajana. Konttiteknologiassa yhteen konttiin pakataan applikaatio sekä sen ajoympäristö helposti siirreltävään muotoon. Yksittäiset kontit voidaan linkittää yhteen muiden konttien kanssa, jolloin kontit voivat kommunikoida keskenään. (Liu ym., 2020). Konttiteknologiaa voidaan kuvailla yhtenä devOpsin mahdollistavana teknologiana, sillä se perustuu infrastruktuuriin liittyvien hallinnollisten töiden ja monitorointitöiden automatisointiin. Kontit mahdollistavat nopean kääntämisen ja iteroinnin, joka tukee devOps-putken jatkuvaa kääntämistä. (Kousa ym., 2020).

Pilvinatiivit sovellukset ovat rakennettu mikropalvelumallin periaatteiden mukaisesti, ja ne on suunniteltu erityisesti toimimaan pilviympäristöissä. Pilvinatiiveille sovelluksille tyypillisiksi piirteiksi voidaan luetella ajo automaatioalustoilla sekä migraatio- ja poikkitoimivuusvaatimusten huomiointi. Pilvinatiivit sovellukset toimivat yleensä devOpsin kanssa sujuvasti yhdessä, sillä pilvinatiivit sovellukset on rakennettu devOps-periaatteet huomioiden. (Kratzke & Quint, 2017).

4 Tutkimusmetodologia

Tutkimus tehtiin kvalitatiivisena tutkimuksena. Kvalitatiivisen tutkimuksen tarkoituksena on ymmärtää tutkittavaa asiaa kokonaisvaltaisesti ihmisten kokemuksien kautta (Merriam & Tisdell, 2015). Kvalitatiivinen tutkimus sopii tutkimuskohteeseen, sillä sekä devOps että ohjelmistoarkkitehtuuri ovat liikkuvia konsepteja, eikä niiden implementoimiseen ole yhtä ainoaa kaavaa, joka sopisi kaikkiin ohjelmistoprojekteihin. Kvalitatiivisella tutkimuksella voidaan huomioida kattavammin erilaiset lähtökohdat, joista devOpsia on lähdetty implementoimaan.

Tutkimuksen metodologiana oli tapaustutkimus. Eriksson & Koistinen (2014) määrittelivät tapaustutkimuksen tutkimuksena, jossa *”tarkastellaan yhtä tai useampaa tapausta, joiden määrittely, analysointi ja ratkaisu on tapaustutkimuksen keskeisin tavoite.”* Tutkimusmetodologiaksi valittiin tapaustutkimus, sillä tutkimuksessa projekteja, joissa on työskennelty devOps-kyvykkyyksien parissa, tarkastellaan tapauksina, joka ilmenee eri ohjelmistoissa eri tavalla. Tutkimuksessa tarkoituksena on analysoida devOpsin onnistuneeseen implementaatioon johtaneita arkkitehtuurisia ratkaisuja haastateltujen kertomusten pohjalta. Tapauksia on tässä tutkimuksessa useampia.

4.1 Datan keruu

Tutkimuksessa empiirisen aineiston keruussa käytetään puolistrukturoituja haastatteluja. Puolistrukturoiduissa haastatteluissa haastattelijan tehtävänä on tutustua tutkittavaan aiheeseen ennalta, jonka pohjalta tutkija laatii kysymykset ennen haastattelua. Kysymyksien tarkoituksena on ohjata keskustelua, mutta kysymyspatteria voi mukauttaa haastatteluiden edetessä. Haastatteluiden tarkoituksena on kerätä tietoa samoista aihepiireistä eri haastateltavilta. Kysymyspatteristo suunnitellaan usein pääteemoittain, joista johdetaan jatkokysymyksiä. (Kallio ym., 2016). Puolistrukturoidut haastattelut sopivat tähän tutkimukseen, sillä se antoi haastattelijalle vapauden tarkentaa epäselviksi jääneitä asioita, sekä haastateltavalle vapauden kertoa omin sanoin kokemuksistaan. Lisäksi avoin struktuuri mahdollisti keskustelun keskittymisen jokaisen haastateltavan omaan projektiin sekä siinä ilmenneisiin onnistumisiin ja haasteisiin, kun kysymyksiä voitiin johtaa aina keskustelun edetessä.

Kysymyspatteristo esitettiin devOps-projekteissa työskennelleille arkkitehdeille. Haastateltavat työskentelivät konsulttiyrityksessä, jolloin haastattelumateriaalia saatiin laajasti eri projekteista eri toimialoilta ja ohjelmistoista. Haastateltavat valikoituivat henkilöihin, joilla oli kokemusta sekä ohjelmistoarkkitehtuurista että devOpsista useamman vuoden ja useamman projektin osalta. Haastattelukutsut lähetettiin sähköpostitse ja myöntävän vastauksen jälkeen haastatteluille sovittiin tarkempi ajankohta. Kysymyspatteristo esitettiin suullisena haastatteluna, ja haastattelut äänitettiin puhelimen sekä etäkokoussovelluksen avulla, ja kaikki haastattelut suoritettiin anonyymisti. Haastattelun äänityksen aloitus ja lopetus kommunikoitiin haastateltaville selkeästi. Kysymyspatteristo löytyy tutkimuksen liitteestä A.

Haastattelujen litteroitu versio lähetettiin haastatelluille vielä lopuksi tarkastukseen ennen aineiston hyödyntämistä osana tutkimuksen empiriaa. Haastattelut pidetiin loka-marraskuun vaihteessa 2022 ja haastateltavien tiedot työtehtävistä, kokemusvuosista sekä haastatteluiden päivämäärä ja kesto ovat kerättynä alla olevaan taulukkoon (taulukko 1).

Taulukko 1 Haastattelut

Haastateltava	**Työtehtävä**	**Työkokemus arkkitehtuurien parissa (noin)**	**Haastattelun päivämäärä ja kesto**
H1	EA-Arkkitehti	17 vuotta	24.10.2022 25 min
H2	IT-Arkkitehti	5 vuotta	27.10.2022 20 min
H3	IT-Arkkitehti	15 vuotta	8.11.2022 20 min
H4	IT-Arkkitehti	5 vuotta	10.11.2022 30 min
H5	IT-Arkkitehti	20 vuotta	10.11.2022 35 min

Ennen datan keräämistä datanhallintaa varten luotiin datanhallintasuunnitelma, jonka tarkoituksena on rajata datan hallintaa ja käyttöä sekä muodostaa selkeä suunnitelma siitä, missä ja miten dataa tallenntaan. Datanhallintasuunnitelma löytyy tutkimuksen liitteestä B.

4.2 Datan analysointi

Haastattelut äänitettiin puhelimen äänityssovelluksella sekä etäkokoussovelluksella, ja lopuksi litteroitiin Word-tekstinkäsittelyohjelmalla. Litterointi suoritettiin sanatarkasti, mutta datan analysoinnissa esiin nostetut sitaatit on muutettu puhekielisistä sitaateista kirjakieleen ja puheessa esiintyneet sanojen toistot on karsittu.

Data käsiteltiin koostamalla Excel-taulukkosovellukseen koodit ja niihin liittyvät lainaukset. Koodauksen tarkoituksena on lajitella kerätty kvalitatiivinen data kategorioihin, jolloin tutkimuksesta on mahdollista löytää toistuvuuksia ja yleistyksiä. Koodien luominen voi tapahtua joko datan ulkopuolella, esimerkiksi ottamalla mallia jostain saman tyyppisestä tutkimuksesta, tai sitten koodeja voidaan kerätä datan sisältä. Koodausta suorittaessa voi olla hyödyllistä pitää päiväkirjaa siitä mitä koodeja on kerätty, miksi ne on kerätty ja miten ne istuvat omaan tutkimukseen ja tutkimuskysymykseen. Lisäksi koodausta suorittaessa tutkijan on helpompi löytää yhteneväisyyksiä sekä eroavaisuuksia eri datamassojen välillä, kun tutkijan on mahdollista palata aina omiin muistiinpanoihin datan analysoinnin jälkeen. (Stuckey, 2015). Tässä tutkimuksessa hyödynnettiin koodausta tutkimusmenetelmänä, sillä sen avulla saatiin karsittua tutkimusaineistosta olennainen muutamien avainsanojen avulla. Tutkimuksessa koodit kerättiin datan sisältä ensimmäisen lukukierroksen jälkeen. Datasta kerätyt sitaatit ja koodit kategorioineen ovat löydettävissä tutkimuksen liitteestä C.

Datan analysoinnissa hyödynnettiin Auerbachin ja Silvesteinin (2003) esittelemää mallia. Mallissa on ensin tarkoituksena tehdä tekstistä ymmärrettävää ja tekstin työstämisen jälkeen muodostaa koodatusta aineistosta teoria. Vaikka mallissa olevat työvaiheet ovatkin kuvattuna lineaarisessa järjestyksessä, Auerbach ja Silverstein huomauttavat, että yleensä dataa analysoidessa eri työvaiheiden välillä liikutaan joustavasti ja välillä työvaiheissa voidaan palata taaksepäin. Auerbachin ja Silvesteinin kuvaileman mallin työvaiheet ovat kuvattuna alla:

1. **Tekstin työstäminen**

1.1. Tutkimusaukon ja teoreettisen viitekehyksen kirjaaminen ylös

1.2. Tutkimuskysymyksen kannalta tärkeiden kohtien valikointi esimerkiksi alleviivaamalla

1.3. Toistuvuuksien ryhmittely

1.4. Ryhmien jaottelu kategorioihin

2. Teorian muodostaminen

2.1 Kategorioiden jaottelu abstrakteihin konsepteihin, jotka ovat linjassa viitekehyksen kanssa

2.2 Teoreettisen mallin kokoaminen teoreettisten käsitteiden avulla

Tutkimuksessa lähdettiin kirjallisuuskatsauksen ja tutkimusaukon pohjalta keräämään olennaista tietoa kokoon valikoimalla aineistosta tärkeimpiä kohtia. Valikoinnin jälkeen aineisto ryhmiteltiin toistuvuuksiin, jotka jaettiin vielä useampaan kategoriaan. Kategorioinnin jälkeen aineistoa peilattiin kirjallisuudesta kerätyn teorian kanssa, ja havainnoista koottiin teoreettinen taulukko.

5 Tulokset

Datan analysoinnin jälkeen haastatteluista poimittiin kolme tutkimuksen kannalta tärkeää kategoriaa, onnistumiset, haasteet sekä tehdyt ratkaisut. Tuloksissa esitellään nämä kolme kategoriaa, jaotellen ne vielä kategorioista poimittuihin ja johdettuihin alakategorioihin. Lisäksi tuloksissa esitellään haastatteluissa esitellyt projektit ja projektiohjelmistojen arkkitehtuurit sekä projekteissa hyödynnetyt devOps-työkalut.

5.1 Esitellyt projektit ja arkkitehtuurit

Haastatteluissa haastateltavia pyydettiin esittelemään yksi tai useampia projekteja, joissa oli luotu devOps-kyvykkyyksiä johonkin olemassa olevaan arkkitehtuuriin. Haastatteluissa selvitettiin projektien toimialaa, sekä ohjelmiston arkkitehtuuria, kokoa ja kompleksisuutta. Alla olevassa taulukossa (taulukko 2) on koottuna projektien toimialat, sekä lyhyet kuvaukset ohjelmistoista ja niiden arkkitehtuurista.

Taulukko 2 Projektit

Projekti	**Toimiala**	**Kuvaus**
P1	Julkinen sektori	Monitoimittaja-alustan ylläpitäminen, jossa muun muassa apeja, sovelluksia ja webbisivustoja. Monoliittisesta sovelluksesta on siirrytty konttipohjaiseen alustaan.
P2	Vähittäiskauppa	Vähittäiskaupan integraatioalusta, joka koostuu EAI-komponenteista (sanomajonot) sekä muutamasta virtuaalikoneesta. Siirrytty konttipohjaiseen alustaan.
P3	Logistiikka-ala	Logistiikkajärjestelmän uusiminen, jonka yhteydessä rakennettiin yleiskäyttöisiä työkaluja ohjelmiston integroimiseen, testaamiseen ja julkaisuun. Portaaliohjelmisto.
P4	Kuluttajahyödykkeet	Integraatioalusta, joka koostui usean eri järjestelmän (ERP, HR, verkkokauppa, taloushallinto) välisestä kommunikaatiosta. Yksittäiset pienemmät palvelut.
P5	Utilities-toimiala	Pirstaloituneiden devOps-käytänteiden ja työkalujen yhtenäistäminen sekä devOps-alustan luominen. Ei

		voida varsinaisesti puhua yhdestä ohjelmistoarkkitehtuurista, vaan jokaisella kehitysprojektilla oma lähtökohtansa.
P6	Telekommunikaatio	Valmisohjelmistoon osittain perustuvan ratkaisun toimittaminen, operointimallin luominen ratkaisulle. Java Enterprise Stackiin perustuva SOA-arkkitehtuuri.

Suurin osa uusimmista projekteista oli rakennettu konttipohjaisiksi alustoiksi, kun taas vanhemmat ratkaisut perustuivat enemmän monoliittisiin malleihin sekä portaaliohjelmistoihin. Projektien ohjelmistot sekä toimialat vaihtelivat laajasti, jolloin empiriasta oli mahdollista kerätä laajempaa dataa liittyen erilaisten ohjelmistojen devOps-kyvykkyyksien luomiseen.

Kahdessa projektissa ohjelmistona oli ollut integraatioalusta, jonka tehtävänä on välittää kommunikaatiota eri järjestelmien välillä esimerkiksi hyödyntämällä sanomajonoja.Yhden projektin ohjelmistona oli monitoimittaja-alusta, jossa alustan päällä pyöri useampia sovelluksia ja verkkosivustoja. Alusta ei itsessään ottanut kantaa siihen, miten sen päällä pyöörivät sovellukset on rakennettu. Kaksi vanhemmista esitellyistä projekteista olivat logistiikka-alan portaaliohjelmisto sekä telekommunikaatio-alan valmisohjelmistoon perustuva ratkaisu. Yhdessä projektissa keskityttiin yhden ohjelmiston sijaan devOps-käytänteiden yhdistämiseen, kun yrityskauppojen ja yhdistymisten myötä organisaation devOps-käytänteet olivat pirstaloituneet.

Empriasta oli poimittavissa samankaltaisia havaintoja kuin Liang ym. (2020) oli tehnyt omassa tutkimuksessaan, jossa devOpsin ja arkkitehtuurin suhde nähtiin niin, että mikropalvelumallia tai muuta kompleksista arkkitehtuuria ylläpitääkseen organisaatiolla tulisi olla olemassa devOps-kyvykkyydet. Yksi haastateltavista (H3) nosti esille, että useimmiten devOps-kyvykkyyksien luominen on tapahtunut joko osana pilvitransformaatiota, jossa paikallisesta kehittämisestä on siirrytty kohti pilvipohjaisia alustoja, tai arkkitehtuurimigraatiossa, jossa monoliittisesta mallista on siirrytty kohti mikropalvelupohjaista mallia. Sekä empiriassa että kirjallisuuskatsauksessa on huomioitu arkkitehtuurin ja devOpsin suhde myös niin, että ennen devOps-kyvykkyyksien luomista

on tehty jonkin asteinen arkkitehtuurimigraatio, jossa joko koko mallia tai sen osia on muokattu modulaarisempaan suuntaan.

5.2 DevOps-työkalut

Haastateluissa käytiin läpi projekteissa hyödynnettyjä devOps-työkaluja, ja mainitut devOps-työkalut sekä konttiteknologiaan liittyvät työkalut listattiin kategorioittain alla olevaan taulukkoon (taulukko 3). Taulukon kategorioina ovat devOps-tuotantoputkeen liittyvät työkalut, konttiteknologiaan liittyvät työkalut sekä testityökalut. Jokaisen työkalun tai teknologian vieressä on suluissa kirjattuna, kuinka moni haastateltavista oli hyödyntänyt kyseistä työkalua tai teknologiaa.

Taulukko 3 DevOps-työkalut

DevOps-tuotantoputki	
CI/CD-työkalu	Azure DevOps (5)
	Jenkins (4)
	Nexus (1)
	IBM Continuous delivery (1)
	AWS Code Commit (1)
Versionhallinta	Git (3)
	Bitbucket (1)
Konttiteknologia	
Konttialusta	OpenShift (3)
Konttirekiseri	Quay (2)
Konfiguraatiohallinta	ConfigMap (OpenShift) (1)
Salaisuuksienhallinta	SecretMap (OpenShift) (1)
	Azure KeyVault (1)
Testityökalut	
Testiautomaatio	Robot Framework (2)
	Selenium (1)
	SonarQube (1)
Tietoturvatestaus	
	Owas Zap (1)
	Snyk (1)

Suorituskyvyntestaus	JMeter (2)

DevOps-tuotantoputken työkaluista Azure DevOps oli kaikista käytetyin, sillä kaikki haastateltavat vastasivat hyödyntäneensä Azure DevOpsia jossain projektissa. Toinen usein mainittu devOps-työkalu oli Jenkins, josta mainitsivat neljä haastateltavista, toisin lähes kaikissa tapauksissa Jenkinsiä hyödynnettiin lähinnä vanhemmissa tai pienemmissä projekteissa, kun taas uudemmissa projekteissa Azure DevOps oli yleisempi. Muita mainittuja CI/CD-teknologioita olivat Nexus, IBM Continuous Delivery ja AWS Code Commit. DevOps-työkaluista versionhallintajärjestelmistä Git mainittiin kolmessa eri haastattelussa, ja se oli versionhallintajärjestelmistä selkeästi yleisin, Bitbucket mainittiin yhdessä haastattelussa.

Kolme osallistujista olivat hyödyntäneet konttiteknologioita projekteissaan, ja konttialustoista OpenShift-konttialusta mainittiin kaikissa kolmessa haastattelussa. OpenShiftin ominaisuuksista mainittiin ConfigMap ja SecretMap, jotka oli koettu devOps-työskentelyn kannalta hyödyllisiksi. Lisäksi konttirekistereistä Quay mainittiin kahdessa eri haastattelussa ja Azure KeyVault mainittiin yhdessä haastattelussa.

Testityökaluista Robot Framework mainittiin kaksi kertaa ja Robot Frameworkin lisäksi Selenium ja SonarQube mainittiin kahdessa eri haastattelussa. Tietoturvatestauksen työkaluista Snyk ja Owas Zap olivat käytössä yhdessä projektissa, ja suorituskykytestaustyökalu JMeter mainittiin kahdessa eri haastattelussa.

5.3 Haasteet

Haastatteluissa selvitettiin projekteissa esiin nousseita haasteita devOpsin ja arkkitehtuurin yhteensovittamisessa. Datan koodauksen jälkeen haasteet voitiin jaotella kahteen eri kategoriaan, teknologisiin ja organisatorisiin haasteisiin. Organisatorisista haasteista pystyttiin tunnistamaan henkilöstöön sekä talouteen liittyviä haasteita, ja teknologiset haasteet ryhmiteltiin vielä tarkemmin vanhojen teknologisten ratkaisuiden haasteisiin, joka sisälsi sekä työkaluja että arkkitehtuurisia malleja, alustan sopivuuteen sekä työkalujen valintoihin. Datasta löydettyjä kategorioita havainnollistetaan taulukossa 4.

Taulukko 4 Haasteet

Haasteet				
Teknologia			Organisatoriset	
Työkalujen valinnat	Vanhat teknologiset ratkaisut	Alustan sopivuus	Henkilöstö	Taloudelliset
Toimivan softapaletin löytäminen	Manuaaliset asennukset Konfiguraatioiden hallinta Riippuvuuksien hallinta Monoliittinen malli	Eristetyt on-premises järjestelmät Erilliset ohjelmointikielet Pakettisofta	Muutoksen hallinta Osaamisen erillisyys	Kustannukset Kustannusoptimointi

5.3.1 Vanhat teknologiat

Bolscher & Daneva (2019) totesivat artikkelissaan, että usein devOpsin implementoinnissa haasteita aiheuttaa tiukasti toisiinsa liitoksissa oleva monoliittinen arkkitehtuuri. Myös haastatteluissa monoliittinen malli nostettiin esille devOpsin kanssa heikommin yhteensopivana mallina (H4).

> *"Jos on just tämmöinen yksi iso monoliitti tai jos on tosi pieni tiimi niin se hyöty jää helposti pienemmäksi ja monet asiat voi olla nopeampi tehdä vaan käsin." (H4).*

Koska monoliittinen malli perustuu siihen, että koko ohjelmisto nivotaan yhteen käännökseen, soveltuu se heikommin jatkuvan kääntämisen periaatteisiin kuin modulaarisempi malli (Astudillo ym., 2019). Sama havainto nostettiin esiin haastatteluissa. DevOpsin tarkoituksena on automatisoida eri tuotantovaiheita mutta haastatteluissa koettiin, että monoliittisen mallin automatisoinnin hyöty oli jäänyt pieneksi, ja siksi monoliittisessa mallissa manuaalinen työ koettiin nopeammaksi tavaksi toteuttaa työskentelyä.

Järjestelmäarkkitehtuurien haasteena on usein, että se jää jälkeen ketterien kehitystapojen kehityksestä (Ravichandran & Waterhouse, 2016) ja myös haastatteluissa

nostettiin esille vanhojen teknologioiden tuomat haasteet devOps-kyvykkyyksien luomisessa (H1, H2, H5). Esimerkiksi vanhojen teknologioiden päälle rakennettuja ratkaisuja ei usein olla suunniteltu toimimaan osana devOps-putkea ja osa alustoista ei tue devOps-käytänteiden ja työkalujen hyödyntämistä, jolloin devOpsista saatavat hyödyt jäävät myös selkeästi pienemmiksi.

> *”Haasteet on ollut välillä teknologiaan liittyviä, että kaikki tämmöiset alustat ja platformit ei välttämättä tue kauhean hyvin devOps-käytänteiden ja työkalujen hyödyntämistä. Paljon on käytössä ohjelmistoja edelleen, joiden ajatusmalli perustuu vähän vanhempaan ajatteluun ja niitä on vaikea saada sovitettua tämmöiseen devOps-malliin.” (H5)*

Haastattelussa esiin nostettiin ongelma siitä, miten devOps-käytänteet saadaan asetettua arkkitehtuuriin silloin, kun ohjelmistojen rakenne ei ole siihen sopiva. Erityisesti monoliittisten järjestelmien haasteena on historian saatossa tehdyt ratkaisut ja valinnat, jotka eivät vastaa enää nykyisen liiketoiminnan tarpeisiin (Ravichandran & Waterhouse, 2016). Empiriassa tämä nousi esille ratkaisuina, joita ei ole suunniteltu vastaamaan devOps-tyyppisen kehityksen vaatimuksiin, jolloin myös devOps-työkalujen tuominen osaksi kehittämistä on koettu haastavana.

Mishra & Otaiwi (2020) nostivat esille jatkuvan kääntämisen ja arkkitehtuurin välisen yhteistyön. Yhtenä esimerkkinä vanhempaan teknologiaan liittyvistä ongelmista on koodin automaattisen kääntämisen suorittaminen, ja haastatteluissa nostettiin esille, että vanhemmissa järjestelmissä esimerkiksi konfiguraatioita oli tyypillisesti asennettu käsin, jolloin sovellusten paketoinnista löydettiin puutteita, ja vajavaisen datan takia sovelluksen asettaminen osaksi devOps-putkea oli käytännössä mahdotonta (H1, H2).

> *”Se on historiallisista syistä asennettu jollakin skriptillä ja muulla ja sitten siihen on asetettu konfiguraatiota klipsuttelemalla sitä järjestelmää eli että me haluttaisikin deployata se koodi ja me haluttaisiin deployata ne konfiguraatiot automaattisesti, niin semmoiset on vaikeita. Ainakin näihin olen itse törmännyt.” (H1.*

> *”No ehkä silloin ensimmäisessä projekteissa ongelma oli vähän se, että devOpsissa tämä deployment, joka on sen ihan ensimmäinen vaihe oikeastaan, niin ne sovellukset, kun ne oli vielä alun perin asetettu käsin niin konfiguraatio ei ollut kovin hyvin hallussa eli nyt piti ensin tavallaan kerätä ja paketoida se konfiguraatio, jotta sitten näitä devOpsin työkaluja pystyi ylipäätään käyttämään.” (H2)*

Mikropalvelumallissa konfiguraatiokerros voidaan joissain tapauksissa välttää hyödyntämällä konttiteknologiaa, jossa kuvarepositoriosta (engl. image repository)

voidaan noutaa valmiita kuvia, joita voidaan muokata vain tarvittaessa (Balalaie ym., 2018). Konttiteknologialle ja kuvarepositorioille vaihtoehtoisena tai joissain tapauksissa täydentävänä ratkaisuna voidaan pitää konfiguraatioidenhallintaa (Leite ym., 2019). Haastateltavien mukaan konfiguraatioidenhallinnassa haasteellisiksi ovat nousseet sellaiset tapaukset, joissa konfiguraatioidenhallinta on ollut aiemmin puutteellista ja joissa konfiguraatioidenhallinnan automatisointia ei ole lähtökohtaisesti otettu huomioon kun konfiguraatioita on tehty. Tällaisissa tapauksissa konfiguraatiot on jouduttu ensin keräämään ja paketoimaan ennen kuin jatkuvaa kääntämistä on voitu toteuttaa.

Mikropalveluarkkitehtuurissa haasteeksi muodostuu usein riippuvuuksien hallinta, sillä mikroplavelumallin rakenne on monoliittista mallia huomattavasti kompleksisempi (Astudillo ym., 2019). Myös haastatteluissa nostettiin esiin riippuvuuksiin liittyvät ongelmat (H2).

> *”Ne on kuitenkin aika monimutkaisia ne build-konfiguraatiot niissä sovelluksissa, sitten niiden riippuvuudet erilaisissa kirjastoissa on aika monimutkaisia” (H2).*

Haastateltavan tapauksessa devOps-putkesta oli jätetty ulkopuolelle sovellusten paketoinnin automatisointi, sillä paketoinnin tuottaminen osaksi devOps-putkea oli osoittautunut liian työlääksi. Tässä tapauksessa haasteita aiheutti erityisesti sovellusten riippuvuudet ja niiden hallinta, joka olisi tehnyt paketoinnista ja konfiguraatioista monimutkaisia. Bolscher & Daneva (2019) nostivat tutkimuksessaan esille riippuvuuksiin liittyvät ongelmat, jotka voivat estää monoliittisen järjestelmän purkamisen komponentteihin. Vaikka arkkitehtuurimallin muuttaminen modulaarisemmaksi onkin devOps-toimintamallin kannalta järkevää, voi modulaarisuus aiheuttaa uusia haasteita riippuvuuksien ja niiden hallinnan kanssa. Tällöin ratkaistavaksi jää löytää sellainen malli, johon devOps-kyvykkyydet saadaan luotua ilman, että riippuvuuksienhallinnasta muodostuu liian työlästä.

5.3.2 Alustan sopivuus

DevOps-implementoinnissa alustan sopivuus voi aiheuttaa haasteita devOpsin ja arkkitehtuurin yhteensovittamisessa. Datan analysoinnin pohjalta haasteiksi nousivat ratkaisut, joissa devOps joudutaan implementoimaan erilaisilla kustomoiduilla ratkaisuilla valmiiden ratkaisuiden sijaan (H3, H4). Esimerkiksi arkkitehtuureista haasteellisina nostettiin esiin eristetyt on-premise järjestelmät, jotka eivät sovi valmiisiin

ja olemassa oleviin viitekehyksiin, vaan devOpsin implementoinnissa joudutaan tekemään omat ratkaisut. On-premise -järjestelmä on järjestelmä, joka pyörii yrityksen omissa koneissa ja palvelimissa ja usein sen ylläpidosta vastaa yrityksen IT-henkilöstö (Meghazani, 2019). Myös erilliset ohjelmointikielet nostettiin esille devOpsin automatisointia hankaloittavana ratkaisuna.

> *”On jouduttu tekee jotain eristettyjä on-premise järjestelmiä, niin niissä on tavallaan jouduttu keksiin pyörä uudestaan, niin nämä olen nähnyt aina haasteellisina” (H3).*

> *”Siinä on oma ohjelmointikielensä, niin se on aika hankalaa ensinnäkin paketoida ne muutokset ja integraatiot mitä tehdään ja konfiguroida ne ja testata. Se vaatii aika paljon käsityötä.” (H4).*

Ohjelmointikieli sekä järjestelmän alusta ovat arkkitehtuurisia ratkaisuja, jotka vaikuttavat devOpsin automatisoinnin onnistumiseen, ja näissä esimerkeissä on-premise -järjestelmän ja erillisten ohjelmointikielien haasteeksi muodostuu niiden yhteensopimattomuus devOpsin valmiiden työkalujen kanssa. Esimerkiksi Microsoft on tehnyt dokumentaation ainoastaan muutamalle tunnetulle kielelle applikaatioiden paketointiin liittyen sekä ainoastaan muutamalle alustalle, joille applikaatioita voidaan kääntää (Mullans & Shaub, 2018). Koska devOps-työkalut on rakennettu sopimaan tietyille ohjelmointikielille ja alustoille, voivat näiden kielten ulkopuoliset implementaatiot olla hyvin hankalia sovitettavia devOps-käytänteisiin. Haastatteluissa ohjelmointikielen erillisyys oli johtanut siihen, että suurin osa automatisoinneista jouduttiin luomaan käsin, jolloin devOpsin automatisoinnin hyödyt jäävät huomattavasti pienemmiksi.

Yksi alustan sopivuuden haasteista liittyy myös devOpsin kokonaisvaltaisuuteen, jossa pelkästään yhden osan sopivuus ei vielä takaa koko alustan sopivuutta. On esimerkiksi huomattu, että vaikka jokin alustassa tehty ratkaisu sopisikin devOps-tyyppiseen kehittämiseen, voi järjestelmän arkkitehtuuri aiheuttaa edelleen huomattavia haasteita devOps-kyvykkyyksien luomisessa. Vaikka pääsääntöisesti esimerkiksi pilvipalveluiden ajatellaan sopivan hyvin devOps-tyyppisen kehittämisen kanssa (Kraztke & Quint, 2017), niin Lkwatare ym. (2019) nostivat tutkimuksessaan esiin, että esimerkiksi julkisen pilven käyttäminen voi aiheuttaa huomattavia haasteita devOps-implementaatiossa vastuukysymyksien vuoksi. Myös haastatteluissa nostettiin esiin, että pelkästään pilvessä tarjottu palMullvelu ei ole tae alustan sopivuudesta (H5).

> *“Se missä se infra on, on se sitten pilvessä tai on-premissä niin se ei sinänsä vielä kerro sitä kokonaiskuvaa siitä, että miten hyvin devOps-toimintamalli istuu sen ratkaisun ympärille." (H5)*

Haastattelussa viitattiin jo aiemminkin esiin nostettuihin toiminnanohjausjärjestelmiin, joiden soveltuvuus devOps-tyyppiseen kehittämiseen on heikkoa, vaikka järjestelmä olisikin pilvipohjainen ja sen perusteella voisi olla yhteensopiva devOps-tyyppisen kehittämisen kanssa. Ravichandra & Waterhouse (2016) nostivat esille, että devOpsissa oleellista on ajatella järjestelmiä kokonaisuuksina. Kun järjestelmälle lähdetään luomaan devOps-kyvykkyyksiä, tulisi järjestelmässä huomioida sekä infrastruktuurin sijainti, järjestelmän arkkitehtuuri, että näihin soveltuvat työvälineet. Jos jokin näistä ratkaisuista ei sovi devOps-työmalliin, voi se aiheuttaa huomattavia haasteita devOps-tyyppisesssä työskentelyssä.

Pakettisoftalla viitataan johonkin valmiiseen ohjelmistoon, jonka yritys ostaa tukemaan jotain toimintoaan ja yksi esimerkki pakettiohjelmistoista on toiminnanohjausjärjestelmä SAP (Light, 2005). DevOps-kyvykkyyksien luominen pakettisoftaan on koettu empirian perusteella haastavaksi, ja muutamat haastateltavat nostivat pakettisoftan työläänä devOps-implementaation kannalta (H1, H5).

> *”Periaatteeseen ongelma on usein siinä, että kun on pakettisofta niin sitä ei olekaan mietitty sillä tavalla, että tämä tulee devOpsin kautta asetettuna.” (H1)*

Pakettisoftan haasteena nostettiin empiriassa esiin sen yhteensopimattomuus devOps-käytäntöjen kanssa. Light (2005) esitti, että vaikka pakettisoftan tarkoituksena on usein korvata jokin jo olemassa oleva vanha teknologia (engl. legacy application) niin tosiasiassa pakettisoftat on jo itsessään rakennettu hyödyntämään vanhoja teknologioita. Hänen mukaansa pakettisoftan ongelma on usein se, että pakettisofta ei välttämättä tue kaikkia niitä toimintoja, joita asiakas haluaisi softan avulla toteuttaa. Tämä ongelma nostettiin esiin myös haastatteluissa, sillä pakettisoftat eivät tukeneet olemassa olevia devOps-työkaluja ja käytäntöjä ja devOpsin implementoiminen tällaiseen ohjelmistoon on haastavaa.

5.3.3 Työkalujen valinnat

Ravichandran ja Waterhouse (2016) esittivät, että toimivan ohjelmistoarkkitehtuurin saavuttamiseksi arkkitehtien tulisi kommunikoida kehittäjien kanssa teknologisista

valinnoista, joita devOps-ratkaisussa ja arkkitehtuurissa halutaan käyttää, sekä pohtia valintojen vaikutusta kokonaiskuvaan. Haastatteluissa esiin nostettiin haaste toimivan teknologian löytämisestä sekä sen osalta että se sopii kyseiseen projektiin ja arkkitehtuuriin, että sen osalta että tiimillä on tarvittavaa osaamista hyödyntää valittuja teknologioita (H1, H4).

> *”Että just se, että löydät semmoisen softapaletin mikä toimii, koska se on kuitenkin vähän ehkä testattu silleen olettaen, että tämä vois olla hyvä ja sitten kun sitä ruvetaan käyttämään, niin havaitaankin että tämä ei ollutkaan se paras vaihtoehto. Että se valinta on vaikea.” (H1)*

> *”Ehkä se vaikeus on siinä, että on paljon eri vaihtoehtoja ja eri ihmisillä on eri käsitykset asioista, eri kokemuksia, eri mieltymyksiä niin sitten semmoisen hyvän ja selkeän mallin löytäminen voi olla tosi vaikeata.” (H4)*

DevOpsin ympärille on rakennettu paljon erilaisia teknologioita ja tekniikoita, kuten esimerkiksi versionhallintajärjestelmiä, CI/CD-putkia ja testiautomaatioratkaisuja, ja usein sopivan ratkaisun löytäminen kaikista vaihtoehdoista voi olla haastavaa. Joskus ratkaisun haasteet saattavat ilmetä vasta ratkaisun käyttövaiheessa, jolloin valinnaksi jää joko korjata ratkaisun haasteet, sietää niitä tai vaihtaa teknologiaa. Leite ym. (2019) nostivatkin esille sen, että devOps-työkalujen tulisi sopia tiimin olemassa olevaan osaamiseen, jotta niistä on mahdollista saada hyötyä irti. DevOps-työkalujen lisäksi implementoitavan arkkitehtuurimallin tulisi olla sellainen, että organisaatiossa on riittävää osaamista sen hallitsemiseen, ja esimerkiksi Babar ym. (2021) nostivat esille mikropalvelumallin haasteen sen kompleksisen rakenteen vuoksi, jolloin mallin ylläpitäminen vaatii myös osaavia kehittäjiä ja asiantuntijoita. Jos järjestelmän kehityksessä on mukana useita eri tiimejä, jotka ovat tottuneet toimimaan tietyillä alustoilla ja teknologioilla, voi kaikille sopivan ratkaisun löytäminen olla haastavaa. Tiimien osaamisen lisäksi valittujen teknologioiden pitäisi olla kohtuullisella työmäärällä toteutettavissa ohjelmiston rakenteeseen, joka oli empiriassa osoittautunut haasteelliseksi.

5.3.4 Henkilöstö

Ravichandran ja Waterhouse (2016) nostivat artikkelissaan esiin devOpsin monitahoisuuden, ja devOpsia implementoidessaan yritys joutuu myös miettimään muutoksia organisaatiorakenteessa. Haastatteluissa esiin nousevat haasteet liittyivät myös muutoksen hallintaan, sillä usein tiimeillä on entuudestaan tuttuja teknologioita ja

tapoja toimia, joita joudutaan mahdollisesti muuttamaan kun devOps otetaan käyttöön (H1, H3, H4).

> *”Jos se tuleekin projektilla ilman osaamista siihen, tai että joku projekti on, että heillä on tiimi, joka sanoo, että he ovat paljon tehokkaampia, jos saa käyttää jotain toista teknologiaa. Mutta se on enemmän ehkä sen muutoksen hallintaa tai sitten organisaation hallitsemista, että ne työkalut ovat semmoisia, että ne kelpaavat ihmisille.” (H1).*

DevOps-työkalujen käyttöönotossa muutosjohtaminen on tärkeää, jotta valitut ratkaisut saadaan sopimaan tiimin työskentelytapoihin. Ravichandran ja Waterhouse (2016) nostivat myös esille johtamisen tärkeyden osana devOpsin käyttöönottoa, jotta tiimillä on mahdollisuus kommunikoida sujuvasti keskenään, sekä luoda yhteistä työskentelytapaa. Myös haastatteluissa nostettiin esille johtamisen tärkeys, sillä pelkästään uusien työkalujen käyttöönotto ei riitä luomaan onnistunutta devOps-kulttuuria, vaan työkalut pitäisi pystyä tuomaan sujuvasti osaksi tiimin päivittäistä työskentelyä niin että niiden käyttöön on riittävää osaamista tai halua opetella uusia työvälineitä.

Yksi huolenaiheista oli myös osaamien erillisyys, sillä vaikka arkkitehtuuri ja devOps voidaan ajatella kahtena erillisenä osa-alueena, ne ovat tiiviisti nivoutuneina yhteen (H3). Haastatteluissa nostettiin esille se, että usein devOps- ja arkkitehtitiimit toimivat toisistaan erillään, ja usein arkkitehtitiimissä toimivilla henkilöillä ei välttämättä ole riittävää osaamista devOpsista käytännön tasolla.

> *”Koska selkeästi tavallaan ne asiat mitä löydettiin käytännön kautta devops-pipeline työstä, niin ne heijastuivat siihen arkkitehtityöhön mutta ne ihmiset olivat kuitenkin eri ihmisiä.” (H3).*

Osaamisen jakautuminen kahteen eri tiimiin aiheuttaa haasteita, jos tiimien välinen kommunikaatio ei ole riittävällä tasolla. Yhtenä sekä devOpsin että mikropalvelumallin yhteneväisenä piirteenä on nostettu esiin poikkiosaavien tiimien hyödyntäminen työskentelyssä (Liang y.m. 2020) ja esimerkiksi toimivan devOps-mallin implementointi vaatii arkkitehtien ja kehittäjien välistä tiivistä kommunikaatiota (Ravichandran & Waterhouse, 2016). Sekä kirjallisuus että empiria nostavat esiin huolen siitä, miten arkkitehtuuriset ratkaisut voidaan sovittaa devOps-tiimin osaamiseen, kun tiimit toimivat toisistaan erillään eivätkä välttämättä tunne toistensa työtapoja riittävästi.

5.3.5 Taloudelliset

Leite ym. (2019) mukaan uusien devOps-työkalujen käyttöönotossa tulisi aina huomioida työkaluista saatava mahdollinen hyöty suhteessa niiden käyttöönotosta aiheutuviin kustannuksiin. Yhtenä devOps-ratkaisuiden haasteena voidaan pitää kustannuksia, joita devOps-työkalujen ja ratkaisujen implementointi aiheuttaa. DevOps-työkaluja voidaan joutua jättämään ulos niiden kustannusten takia ja devOps-työkalujen valinnassa joudutaan usein priorisoimaan valittuja ratkaisuja (H1, H4).

> *”Minä sanoisin, että ensisijainen syy, että joku jätetään ulos, on ehkä enemmänkin kustannukset, kun niinkään arkkitehtuuri. Ne kuitenkin voi olla hyviä sovelluksia mutta sitten hinta tai hankinta on vaikeaa ja siksi ne jäävät ehkä pois.” (H1).*

Toinen haaste kustannuksiin liittyen on kustannusoptimoinnin sijoittaminen oikeaan hetkeen. Haastatteluissa nostettiin esiin ongelmat, jotka aiheutuvat siitä, kun jo devOpsin implementointi vaiheessa pyritään tekemään asioita kustannusoptimoidusti (H3).

> *”Niiden ongelmat ovat tämmöisiä esimerkiksi, että bisnekseen tulee jotain kustannusoptimointitavoitteita, jotka tavallaan tuhoavat esimerkiksi, että jos kvartaalibonukset liittyvät siihen, että saadaan kustannuksia alaspäin eikä siihen, että saadaan toimiva devOps, joka vastaa tähän arkkitehtuuripäätökseen, niin tämä on tuhoisa tie.” (H3).*

Toimivan devOpsin implementoiminen arkkitehtuuriin vaatii aina organisaatiolta investointeja. Haastatteluissa esitettiin, että kustannusoptimointia voidaan tehdä siinä vaiheessa, kun toimiva devOps on saatu koottua, mutta implementointi vaiheessa toimivan tiimin kasaaminen on välttämätöntä, jotta ratkaisusta tulee hyvä ja toimiva. Lkwatare ym. (2019) tutkimuksessa nostettiin esille, että usein ennalta määritelty hinta ja aika projektille koettiin haasteellisiksi, sillä näihin tavoitteisiin pääseminen johti esimerkiksi testauksen supistumiseen tai ominaisuuksien hätäiseen priorisointiin. Molemmissa tapauksissa ratkaisun laatu huononi selkeästi. Kustannuksia pohdittaessa haasteita tulee sekä siitä, mitä ratkaisuja voidaan ylipäänsä toteuttaa, että siitä, missä kohdassa investointia voidaan alkaa tekemään kustannusoptimointia, jotta se ei vaaranna ratkaisun onnistumista.

5.4 Onnistumiset

Haastatteluissa selvitettiin myös, millaiset asiat arkkitehtuurissa on koettu toimiviksi devOpsin kanssa. Onnistumiset jaettiin haasteiden tapaan organisatorisiin ja teknologisiin

kategorioihin, ja teknologiset onnistumiset jaettiin vielä modulaarisuuteen, alustan sopivuuteen sekä työkaluihin. Organisatorisista onnistumisista oli löydettävissä henkilöstöön liittyviä asioita. Onnistumisia on havainnollistettu taulukossa 5.

Taulukko 5 Onnistumiset

Onnistumiset			
Teknologia			Organisatoriset
Modulaarisuus	Työkalut	Alustan sopivuus	Henkilöstö
Palvelun palastelu Löyhät riippuvuudet Mikropalvelumalli Pilvipalvelut	Pilvinatiivit sovellukset Konttiteknologia	Konfiguraatioiden yhtenäisyys Alustan yhtenäisyys Räätälöity softa	Osaava tiimi

5.4.1 Modulaarisuus

Bolscherin ja Danevan (2019) tekemässä tutkimuksessa devOpsia tukevaksi arkkitehtuurimalliksi nostettiin mikropalvelumalli ja myös haastatteluissa mikropalvelumallin pienet palvelut nostettiin esille devOpsia tukevana piirteenä (H3, H4). Yhtenä ratkaisevana asiana mikropalvelumallissa nostettiin esiin palvelun modulaarisuuden onnistuminen, eli mitä sopivamman kokoisiin ja yhteneviin osiin palvelu oli saatu purettua, sitä helpompaa järjestelmässä oli toteuttaa devOps-toimintoja.

> *”No kyllä minun mielestäni juuri tämmöiset mikropalvelut, pienet palvelut, jos ne ovat saman tyyppisiä vielä, että sama malli toimii niille kaikille komponenteille, niin niistä on paljon hyötyä” (H4).*

Mikropalvelumallin eduiksi devOpsin yhteensovittamisessa on nostettu niiden useammat yhteneväiset piirteet, joista yksi on isojen kokonaisuuksien pilkkominen pienempiin osiin (Liang ym. 2020). Mikropalvelumallin modulaarisuus mahdollista jatkuvan kääntämisen toteuttamisen kun järjestelmä ei ole monoliittisen mallin tapaan nivottuna yhteen käännökseen (Astudillo ym. 2019) ja samankaltaisia kokemuksia oli löydettävissä haastatteluissa, joissa mikropalvelut nostettiin esille devOpsia tukevana mallina.

Arkkitehtuurisista malleista voidaan sekä empirian että teorian pohjalta nostaa esille, että käytännön työssä mallit eivät välttämättä noudata yhtä tiettyä arkkitehtuurista mallia. Esimerkiksi mikropalvelumalli on luonteeltaan hyvin kompleksinen ja osa siinä esiintyvistä teknologioista ei välttämättä sovellu suoraan kaikkiin vanhoihin järjestelmiin (Ravichandran & Waterhouse, 2016), jolloin vanhan järjestelmän sovittaminen modulaarisemmaksi voi vaatia erilaisia kompromisseja eri mallien välillä. Myös haastatteluissa onnistuneita malleja kuvailtiin useamman pienemmän ja yksittäisen palvelun kokonaisuuksiksi (H4).

> *”Mikropalveluarkkitehtuuria näissä minun keisseissä on ollut, ei ole ehkä ihan vielä semmoista kovin pitkälle vietyä mikropalveluarkkitehtuuria eli on kyllä yksittäisiä palveluita, mutta ne ei vielä kovinkaan paljoa juttele keskenään tai ehkä kaksi palvelua juttelee keskenään mutta ei ole semmoista kokonaisvaltaista mikropalveluarkkitehtuuria mutta yksittäisiä palveluita on tosi paljon nykyään ja pienempiä palveluita”(H4).*

Arkkitehtuurimalli voi usein muodostua eri mallien piirteistä, kuten haastatteluissa arkkitehtuurimallia kuvailtaa useamman yksittäisen palvelun kokonaisuudeksi, joka ei malliltaan kuitenkaan vastannut kovin pitkälle vietyä mikropalvelumallia. Esimerkiksi Babar & Shahin (2020) tutkimuksessa puhtaan mikropalvelumallin hyödyntämisen sijaan devOps-toimintamallin yhteydessä oli otettu käyttöön arkkitehtuurimalli, joka sisälsi piirteitä sekä mikropalvelumallista että monoliittisesta mallista. Mallin avulla pystyttiin kiertämään mikropalvelumallin haasteita, jotka johtuvat mallin kompleksisuudesta. Myös Shahin ym. (2016) tutkimuksessa mikropalvelumallin haasteiden takia yksi tutkimuksen organisaatioista oli muokannut monoliittisesta applikaatiosta binääriset riippuvuudet usean osan sijaan. Haastattelun tapauksessa mikropalvelumallista oli puolestaan karsittu kommunikaation toteuttamisesta muodostamalla yksinkertaisia kommunikaatioita palveluiden välille.

Täydellisen mikropalvelumallin implementoimien sijaan mallin modulaarisuuden voidaankin ajatella olevan tärkein arkkitehtuurinen ratkaisu devOpsin onnistumisen kannalta (Babar & Shahin, 2020), ja samankaltaisia löydöksiä voidaan nostaa myös haastatteluista esiin (H5). Sekä haastattelussa että Babarin ja Shahinin (2020) tutkimuksessa toimivaan malliin yhdistettiin modulaarisuuden lisäksi löysät riippuvuudet.

> *”Silloin kun ratkaisuja kehitetään tavallaan järkevästi pilkottuna sopiviin kokonaisuuksiin, jotka on riittävän löyhästi toisistaan riippuvia ja kuhunkin*

> *ratkaisun osaan voidaan valita se tarpeen mukaan paras ja sopivin teknologia, silloin myös yleensä nämä devOps-käytänteet ja -periaatteet on toiminut parhaiten.” (H5).*

Löysiin liitoksiin perustuvat arkkitehtuuriratkaisut mahdollistavat devOps-mallin tyyppistä kehittämistä, sillä ratkaisuissa tarkoituksena on muodostaa palveluita, jotka ainoastaan kommunikoivat keskenään ottamatta kantaa toistensa rakenteisiin (Pautasso & Wilde, 2009). Näin löysillä liitoksilla voidaan kiertää monoliittisen arkkitehtuurin ongelmia, jossa osat ovat tiukasti kiinnitettyinä toisiinsa, ja ohjelmiston päivittäminen on sen takia haastavaa. Löysiin liitoksiin perustuvan arkkitehtuurin on sanottu olevan yksi ratkaisevimmista arkkitehtuurisista piirteistä, kun halutaan luoda devOps-kyvykkyyksiä ohjelmistolle (Babar & Shahin, 2020). Samankaltaisia kokemuksia löydettiin haastattelusta.

5.4.2 Työkalut

Mikropalvelumalleissa virtuaalikoneita tehokkaampana ratkaisuna voidaan pitää konttiteknologiaa, sillä virtuaalikoneisiin verrattuna kontit ovat kevyempiä ja helpommin hallittavia (Liu ym.,, 2020). Myös haastatteluissa nostettiin esiin konttiteknologian tuomat edut devOps-mallin implementoinnissa ja suurin osa oli projekteissaan hyödyntänyt OpenShift-konttialustaa (H1, H2, H4).

> *”Konttiteknologia on sellainen, joka helpottaa devOpsin käyttöönottoa ja toisaalta kääntäen sitten devOps, ja jättäytyi muuten mainitsematta, että git-repositorio on toki tämä versionhallintasysteemi mihin nämä kaikki meidän devops-arkkitehtuurimme on perustunut, niin git-repositoriot ja kontit näyttää toimivan hyvin yhteen tällaisten CICD-ratkaisujen kanssa, että nämä ovat tavallaan hyvä pari.” (H2).*

Esimerkissä nostettiin esiin devOps-työkaluista versionhallintajärjestelmän sekä konttien sujuva yhteistyö, joka puolestaan helpotti CI/CD-putken rakentamista. Konttiteknologia on yksi jatkuvan kääntämisen mahdollistavista teknologioista, sillä se perustuu applikaatioiden ajoympäristön automatisointiin ja applikaatioiden nopeaan kääntämiseen (Kousa ym., 2020). Konttiteknologia sekä devOpsin työkalut toimivat sujuvasti yhdessä, sillä ne molemmat perustuvat samankaltaisiin periaatteisiin, kuten töiden automatisointiin sekä ketterään kehittämiseen.

Toinen devOpsin mahdollistava teknologia arkkitehtuureissa on pilvipalvelut. Pilvipalveluiden etuina ovat resurssien helppo jaettavuus ja saatavuus, jolloin resursseja

voidaan jakaa dynaamisesti ja joustavasti (Anand ym., 2022, 26–27). Datan analysoinnin jälkeen myös pilvipalvelut nousivat yhtenä devOps-projekteja edistävänä tekijänä (H1, H3), ja usein esimerkiksi pilvitransformaatioiden yhteydessä organisaatio on ottanut myös devOps-kyvykkyydet käyttöön.

> *”Sitten on hyvin tyypillistä sellainen, että jos on joku pilvitransformaatio, että organisaatio haluaa siirtyä esimerkiksi, että meillä on paikallista kehitystä ollut, mutta sitten ne haluavat siirtää, vaikka yhteen tai kahteen pilveen niin silloin monesti ne alkavat uudistaa myös sitä omaa devops-pipelinea.” (H3).*

DevOps voidaan nähdä organisaatiossa pilvitransformaation mahdollistajana, ja ilman toimivia devOps-kyvykkyyksiä pilveen siirtyminen voi olla huomattavasti haastavampaa. DevOps ja pilvi kulkevat tiiviisti yhdessä, ja esimerkiksi Anand ym. (2022) mukaan pilvipalveluista on eniten hyötyä silloin, kun arkkitehtuuri hyödyntää huolellisesti paketoituja ja suunniteltuja sovelluksia, jotka ovat mikropalvelumallin tyyppisiä. Arkkitehtuuriratkaisut, jotka hyödyntävät pilvipalveluita, helpottavat sovellusten julkaisua ja lisäksi ne pienentävät operointitiimin tarvetta, joka soveltuu hyvin devOps-malleissa toimivaan poikkiosaavaan tiimirakenteeseen (Leite ym., 2019). DevOpsin ja pilvipalveluiden voidaan siis ajatella olevan pari, jotka tukevat toistensa toimintaa johtuen siitä, että niiden toimintaperiaatteet pohjautuvat samankaltaisiin asioihin.

Pilvinatiiveiksi sovelluksiksi kutsutaan sovelluksia, jotka on suunniteltu toimimaan pilvialustoilla (Kartzke & Quint, 2017). Myös haastatteluissa nostettiin esiin pilvinatiivit sovellukset, joiden asentaminen osaksi devOps-ympäristöä on koettu onnistuneeksi (H5).

> *”Sanotaan näin, että pääsääntöisesti tietysti, ja tämä on varmaan varsin ilmeistä, että sellaiset ratkaisut, jotka on suunniteltu tämmöisinä pilvinatiiveina sovelluksina, niin yleensä niissä se arkkitehtuuri on sitten sellainen, että se tukee myös tällaista devOps-tyyppistä kehittämistä.” (H5).*

Pilvinatiivien sovellusten etuna devOps-käytänteisiin yhdistämisessä on se, että niiden arkkitehtuuri on tehty devOps-työskentelyä tukevaksi jo itsessään. Pilvinatiivit sovellukset hyödyntävät modulaarisuutta, löysiä liitoksia sekä mikropalvelumallin mukaista arkkitehtuuria, ja lisäksi ne pyrkivät alustariippumattomuuteen ja joustavuuteen (Anand ym., 2022). Pilvinatiivit sovellukset hyödyntävät siis useita sellaisia arkkitehtuurisia piirteitä, joita devOps-mallin mukaiset käytänteet ja työkalut vaativat, ja ne mahdollistavat näin ketterää kehittämistä.

5.4.3 Alustan sopivuus

Kun haasteiksi nostettiin empiriassa valmiit pakettisoftat niin vastaavasti devOpsin kannalta onnistuneiksi ratkaisuiksi nostettiin räätälöidyt softat (H1). Räätälöidyissä softissa toimittajalla on parempi mahdollisuus muokata olemassa olevaa ratkaisua devOps-malliin sopivaksi.

> *"DevOps varmaan helpoiten tapahtuu, kun me tehdään jotakin räätälöityä softaa ja jolloin meillä on se kontrolli siitä" (H1).*

Räätälöidyn softan etuna on kontrolli softan muokkaamisesta, jolloin devOps-ratkaisujen ja arkkitehtuurimallin yhteensovittaminen on helpompaa. Kun pakettisoftassa devOps-implementaation ongelmaksi voi muodostua sen yhteensopimattomuus devOps-käytänteiden kanssa ja heikko räätälöitävyys (Light, 2005), voidaan räätälöidyssä softassa tehdä devOpsin kannalta tarvittavia muokkauksia ja tuoda devOps-työvälineitä joustavammin mukaan osaksi kehitystä aina tarpeen vaatiessa.

Räätälöidyn softan lisäksi alustan yhtenäisyys nostettiin esille yhtenä devOpsia tukevana ratkaisuna, sillä yhtenäisyyden avulla esimerkiksi devOps-putkien pystyttäminen oli huomattavasti nopeampaa, kuin mallissa, jossa komponentit on rakennettu hyvin eri tavoilla (H2, H4).

> *"Kun se on sellainen hyvin yhtenäinen alusta, niin on huomattu, että se nopeuttaa paljon, aika nopeasti on saatu devOps-pipelineja sitten uusille konteille pystyyn, eli siellä on ollut huomattavia tehokkuushyötyjä, kun sovellusmassa on melko yhtenäistä niin sitten minun mielestäni se devOpsin tehokkuushyöty on huomattava." (H2).*

DevOps-ratkaisuiden hyödyt nousevat esiin parhaiten, jos valmiiksi olevia ratkaisuja voidaan hyödyntää useilla eri komponenteilla. Yhtenäisen alustan hyötynä on mallin toistettavuudesta saatavat tehokkuushyödyt, kun ratkaisua ei tarvitse räätälöidä jokaiselle komponentille erikseen. Manuaalisissa malleissa ratkaisuja joudutaan rakentamaan käsin aina uudelleen, joka vaatii hyvää dokumentointia ja osaamista, kun taas devOps-putkien ja työkalujen hyödyntäminen on mahdollistanut sen, että valmiita ratkaisuja voidaan ajaa uudestaan ilman ratkaisun uudelleen pystyttämistä. DevOps-ratkaisun uudelleen hyödyntämisen pohjalla täytyy kuitenkin olla sellainen arkkitehtuuri, joka tukee samojen teknologioiden hyödyntämistä. Esimerkiksi Leite ym. (2019) nostivat artikkelissaan esiin, että usein mikropalvelumallin haasteena on mallien heterogeenisyys esimerkiksi

konfiguraatioiden osalta, joten mitä yhteneväisemmät applikaatiot ovat, sitä helpommin ne ovat yhdistettävissä mikropalvelumalliin ja devOps-käytänteisiin.

5.4.4 Henkilöstö

Yhtenä devOpsin tavoitteista on saavuttaa yhteistyötä eri tiimien ja osastojen välillä (Leite ym., 2019). Empiriasta löydettiin myös havaintoja, että mitä osaavampi tiimi on ollut käytössä, sitä toimivampi ratkaisu devOpsista on saatu rakennettua (H3).

> *”Mutta tämä on voittava resepti, että otetaan hyvä tiimi, tehdään uutta, sitten uusi toteutetaan, ja sitten se siirretään sieltä ylläpito-organisaatiolle niin sitä suosittelen vahvasti.”(H3)*

Haastatteluissa painotettiin erityisesti tiimin osaamista devOps-implementaation alussa, ja kun devOps-ratkaisu on saatu riittävällä tasolla toimivaksi, voidaan siirtyä enemmän ylläpitovaiheeseen, jolloin myös tiimin osaamista voidaan muuttaa. Työntekijöiden osaamisen pitäminen ajan tasalla on devOps-työskentelyssä tärkeää, ja devOps-tiimissä kehittäjien on opittava operoivalta puolelta yhtä lailla kuin operoivan puolen on opittava kehittäjiltä (Leite ym., 2019). DevOps-implementointi on yleensä hyvin kokonaisvaltainen prosessi, joka muuttaa paitsi järjestelmäarkkitehtuuria, niin myös organisaation tapoja toimia (Lwkatare ym., 2019). Siksi henkilöstön osaamisen on vastattava tulleisiin muutoksiin niin uusien devOps-työkalujen kuin uuden arkkitehtuurimallinkin osalta, ja mitä osaavamman tiimin kanssa devOps-implementaatiota voidaan lähteä tekemään, sitä onnistuneemmaksi ratkaisut ovat yleensä muodostuneet.

5.5 Ratkaisut

Haastatteluissa osallistujilta selvitettiin, millaisia ratkaisuja devOpsin ja arkkitehtuurin yhteensovittamisessa oli kehitetty. Ratkaisut lajiteltiin onnistumisten ja haasteiden tapaan omiin kategorioihinsa, työkaluihin sekä ratkaisuiden muokkaamiseen. Alla olevassa taulukossa 6 on havainnollistettu ratkaisujen kategoriointia.

Taulukko 6 Ratkaisut

Ratkaisut	
Työkalut	DevOpsin muokkaaminen

Pilvi-agnostiset devOps-työkalut Paketoinnin parantaminen	DevOps-putken osien jättäminen ratkaisun ulkopuolelle Ratkaisun hylkääminen

5.5.1 Työkalut

Konfiguraatioidenhallinnan tarkoituksena on parantaa sovellusten siirreltävyyttä, jolloin sovelluksia voidaan ajaa erilaissa ympäristöissä (Leite ym., 2019). Haastatteluissa haasteina nousivat esiin konfiguraatioiden manuaaliset asennukset, ja ratkaisuna tähän on käytetty sovellusten paketoinnin parantamista (H2).

> *”Silloin aikanaan niitä ratkaistiin ja nimenomaan jos lähdetään näistä ihan ensimmäisistä projekteista missä olen ollut, niin parannettiin niiden sovellusten paketointia jo ihan siellä kehitysvaiheessa, niin että niiden konfiguraatiot olivat paremmin siellä sovelluksen sisällä, jolloin ne pystyttiin tekemään automaattisesti eri ympäristöihin.” (H2)*

Sovellusten konfiguraatioita ja paketointia voidaan parannella, jolloin sovellusten asettaminen osaksi devOps-putkea tulee mahdolliseksi. Konttiteknologiaa ja konfiguraatioidenhallintaa voidaan pitää joko toisiaan täydentävinä tai kilpailevina lähtökohtina (Leite ym., 2019), ja täydentävässä versiossa konfiguraatioidenhallinta voidaan implementoida osaksi konttiteknologiaa esimerkiksi kuvien jakeluun ja muodostamiseen tai konfiguraatioidenhallintaan konttien sisällä (Barua, 2015). Empiriassa konfiguraatioita hallinnoitiin parantamalla sovellusten paketointia, jolloin niiden automatisointi onnistui ja ne voitiin liittää osaksi devOps-putkea.

DevOps-työkalujen implementoinnissa kehittäjillä tulisi olla riittävää osaamista devOps-työkalujen käyttöön (Leite ym., 2019). Yhtenä ratkaisuna devOps-työkalujen ja tiimin osaamisen sovittamiseen oli käytetty pilviagnostisia devOps-työkaluja sekä konttiteknologiaa (H5), jolloin työkaluja voidaan hyödyntää eri alustoilla.

> *”Löydettiin tavallaan tällaisia arkkitehtuuriratkaisuja, että käyttämällä tiettyjä pilviagnostisia devops-työkaluja ja paketoimalla ja jalkautumalla ne konttiteknologian avulla, niin tavallaan saatiin aikaiseksi semmoinen alustaratkaisu joka voi periaatteessa laittaa sitten pystyyn ihan kumpaan tahansa pilviympäristöön ja silti se kehittäjäkokemus niistä työkaluista on suurin piirtein samanlainen ja tavallaan ei ole tarvetta opetella tai tuunata sitä ratkaisua aina sen mukaan että ajetaanko kummassa näistä kahdesta valitusta pilvialustasta.” (H5).*

Tässä tapauksessa konttiteknologiaa on hyödynnetty devOps-työkalujen siirrettävyydessä, jolloin samoja työkaluja voidaan käyttää eri alustoilla, eikä niiden käyttökokemus muutu merkittävästi. Konttiteknologiassa tarkoituksena on paketoida sovellukset niin, että niiden siireltävyys ympäristöstä toiseen on mahdollisimman vaivatonta (Kratzke & Quint, 2019) ja tässä esimerkissä konttien siireltävyys oli yhdistetty erilaisilla pilvialustoilla vaivattomasti toimimiseen, jolloin kehittäjien oli mahdollista hyödyntää työskentelyssään niitä välineitä ja alustoja, joilla he olivat tottuneet operoimaan. Tämä arkkitehtuurinen ratkaisu voi siis tukea esimerkiksi työkalujen valinnasta aiheutuvia haasteita sekä työkalujen sovittamista erilaisille alustoille. Lisäksi ratkaisua voidaan hyödyntää muutosjohtamisen tukena, kun kehittäjien mielipiteitä ja mieltymyksiä voidaan ottaa kattavammin huomioon.

5.5.2 DevOpsin muokkaaminen

Ghantous & Gill (2017) kuvailivat devOpsia automaatiolähtöisenä lähestymistapana ohjelmistokehitykseen, jonka yksi oleellinen osa on automatisoitu tuotantoputki ja jatkuva kääntäminen. Koodin automatisoituihin vaiheisiin voidaan täydellisessä devOps-putkessa lukea koodin paketoiminen, integrointi, kääntäminen sekä testaus. Haastatteluista ilmeni kuitenkin, että joissain tapauksissa devOps-putken osia on jouduttu jättämään ulkopuolelle, sillä tiettyjen vaiheiden automatisointi, kuten testaus tai paketointi, olisi ollut kohtuuttoman työlästä ohjelmiston arkkitehtuurin takia (H2, H3, H5).

> *”No yksi esimerkki, tässä viimeisimmässä projektissa tai integraatioalustaprojektissa niin buildia ei tehdä osana sitä devops-pipelinea vaan buildi tehdään kehittäjien ympäristössä ja sitten gitti repoon viedään valmis sovelluspaketti, jonka konfigurointi sitten vaan otetaan käyttöön siinä devops-putkessa ja sitten tehdään deploymentti että se on yksi ja se olisi ollut työläs toteuttaa mutta ehkä täydellisemmässä ratkaisussa myös build vaihe olisi otettu mukaan siihen CI/CD-putkeen” (H2).*

Vaikka devOps siis teoriassa sisältää ison joukon erilaisia työkaluja ja käytänteitä, niin käytännön tasolla devOps-työvälineet ja käytänteet voivat vaihdella hyvin paljon organisaatiosta toiseen, ja joissain tapauksissa täydellisen devOps-ratkaisun sijaan voidaan jättää joitain devOps-putken osia ratkaisun ulkopuolelle. Leite ym., (2019) huomauttivat devOps-työkaluista, että niiden implementoinnissa pitäisi aina huolellisesti arvioida, mitä hyötyjä niistä on saatavilla suhteessa kustannukseen. Leiten ym. näkemystä tukevat myös haastateltavien (H2, H3, H5) kommentit, että jokainen devOps-putken

vaihe ei ole välttämätön toimivan ratkaisun kannalta ja siksi tärkeintä on, että ratkaisuun mukaan otettavilla vaiheilla saadaan käyttötarve täytettyä, ei niinkään työkalut itsessään.

DevOpsin etuina on nostettu usein esille ohjelmiston laadun paraneminen sekä tuotannonsyklien nopeutuminen, kun työskentelyä voidaan tehostaa erilaisten työkalujen ja menetelmien avulla (Lwakatare ym., 2019). Koska devOps-työkalujen tärkein tavoite on toimia kehittämistä helpottavina teknologioina, on tehtyjä ratkaisuja jouduttu arvioimaan kriittisestikin käytettävyyden kannalta. Empiriasta nousi esiin myös ratkaisu, jossa jo tehty päätös jouduttiinkin perumaan koska sen käyttäminen osoittautui työlääksi (H1).

> *”Tehtiin asiakkaalle Openshift-ympäristö ja he halusivat tietyn sovelluksen mitä olivat käyttäneet ja pilvipalvelun itsellensä omaan ympäristöön ja kun he käyttivät konttiteknologiaa, niin tuntui luonnolliselta, että kun tästä nyt on saatavilla konttiversio, asennetaan se konttiversio. Sitten kun sitä asennettiin, niin havaittiin, että se ei ollut hirveän hyvin tuettu ja dokumentaatio oli heikompaa mutta se saatiin toimimaan. Ongelmat tuli sitten siinä vaiheessa, kun piti ruveta päivittämään, niin tajuttiin että työmäärä mikä joudutaan tekemään, kun tulee uusi versio, että sen tavallaan joutuu käytössä aina käsittelemään, että mitä tässä on muuttunut edelliseen, että me saadaan nämä konfiguraatiot mitä me ollaan tähän openshiftille ja pilveen väännetty, että tämä toimii niin on aika työlästä. Niin todettiin, että no itseasiassa tämä päivittäminen tällä tavalla, kun me kuviteltiin, että päivitys tapahtuu, niin ei siinä ole mitään järkeä, että se on paljon parempi että se heillä on paremmin tuettu virtuaalikoneen versio, käytetään mieluummin sitä että sitten me jouduttiin perumaan päätös ja palaamaan takaisinpäin.” (H1).*

Oikean teknologian ja ratkaisun valitseminen on haastavaa, sillä teknologian toimivuus riippuu paljon myös ratkaisusta, johon teknologia halutaan tuoda. Lwakatare ym. (2019) tutkimuksessa myös päivitys nostettiin yhdeksi haasteeksi joka muodostui infrastruktuurin automatisoinnista. Yksi haasteista liittyi vanhaan tietokantaan, jolloin päivitystä seurasi puutteet datassa. Haastattelun esimerkissä toistui samankaltainen tilanne, jossa päivityksen tekeminen järjestelmään osoittautui ajateltua työläämmäksi. Tässä tapauksessa vaihtoehdoksi muodostui uuden ratkaisun peruminen, sillä aiemmin tehty ratkaisu toimi ympäristössä paremmin.

6 Johtopäätökset

DevOpsin ja arkkitehtuurin suhdetta on tutkittu jo laajasti aiemmassa kirjallisuudessa, ja arkkitehtuurimalleista mikropalvelumalli on nostettu pääsääntöisesti devOps-toimintamallia tukevaksi arkkitehtuuriksi (Balalaie ym., 2016, Bolscher & Daneva 2019). Mikropalvelumalli ja devOps nojaavat molemmat samankaltaisiin periaatteisiin, kuten tiimien poikkiosaavuuteen, joustavaan työskentelyyn sekä kokonaisuuksien pilkkominen pienempiin osiin (Liang ym.,, 2020) ja mikropalvelumallin modulaarisuus tukee devOps-toimintamallin mukaista jatkuvaa kääntämistä ja tuotantoputken automatisointia (Bolscher & Daneva, 2019).

Käytännön työssä arkkitehtuurimallien käyttö on kuitenkin hyvin häilyvää, ja monet ratkaisut sisältävät piirteitä erilaisista malleista. Tällaisia löydöksiä tukivat haastatteluissa esiin nostetut kuvaukset projektien malleista, sekä kirjallisuudessa esimerkiksi Babar & Shahin (2020) tekemä tutkimus. Mallit koostuivat esimerkiksi muutamasta osasta, jotka keskustelivat keskenään, tai ne saattoivat yhdistää monoliittisen mallin sekä mikropalvelumallin piirteitä.

Tutkimuksessa yhteen arkkitehtuurimalliin keskittymisen sijaan lähdettiin tutkimaan yksittäisiä piirteitä arkkitehtuureissa, sekä niiden soveltuvuutta devOps-toimintamallin mukaiseen työhön. Haastatteluista esiin noussut ensimmäinen keskeinen löydös oli, että alustan soveltuvuudella devOps-mallin mukaiseen kehitykseen on merkittävä vaikutus devOps-toimintamallin implementoimiseen. Alustan soveltuvuudesta nostettiin esiin esimerkiksi alustan yhtenäisyys, joka mahdollisti tehokkuushyötyjä, kun devOps-putken osia voidaan uudelleen käyttää kontista toiseen sekä räätälöity softa, joka on helpompi muokata yhteensopivaksi devOps-putken kanssa. Haastavina alustoina voidaan nähdä sellaiset alustat, jotka eivät tue olemassa olevia devOps-käytänteitä tai jotka sisältävät valmiiksi tai vaativat jatkossa paljon manuaalista työtä. Alustan yhteensopivuuden arvioimiseksi voidaan pohtia esimerkiksi seuraavia asioita:

1) Missä alusta sijaitsee: on-premise, pilvi, jokin muu?
2) Millaisia muokkauksia alustaan on mahdollista tehdä?
3) Mitkä ratkaisut alustassa on suunniteltu valmiiksi devOps-toimintamallia tukevina?

4) Onko alustassa yhteneväisyyksiä, joita voidaan hyödyntää devOps-tuotantoputkessa?

5) Miten konfiguraatioidenhallinta on toteutettu?

Toinen keskeinen löydös oli työkalujen valitseminen aina tarpeen mukaisesti, jolloin työkalujen on mahdollista tukea devOps-työtä ilman, että niiden käyttöönottoon kuluu valtavasti resursseja. Työkalujen valinnan tulisi istua paitsi ohjelmiston arkkitehtuuriin, niin myös kehittäjien osaamiseen. Työkaluista ja teknologioista konttiteknologia sekä pilviratkaisut nostettiin esille devOpsia tukevana työvälineenä. Yksi työkalujen valinnassa keskeinen ratkaisu on myös devOps-putken rakentaminen. Täydellisen devOps-putken voidaan ajatella koostuvan testaamiseen, integroimiseen ja kääntämiseen liittyvistä työvaiheista (Ghantous & Gill, 2017), mutta käytännön työssä työvaiheet tulisi aina valita sen mukaan, että ne palvelevat liiketoiminnan tarkoitusta, ja välillä arkkitehtuurisista tai kustannuksiin perustuvista syistä on järkevää jättää osa devOps-putkesta toteuttamatta, tai siirtää toteutusta myöhempään vaiheeseen. Työkalujen valinnassa voidaan nostaa esille seuraavia kysymyksiä:

1) Mitä työkaluja tiimi on käyttänyt aiemmin, millaista osaamista tiimillä on?

2) Mitkä devOps-putken osat tukevat liiketoimintatarpeen täyttämistä?

3) Mitä devOpsia tukevia ratkaisuja arkkitehtuuriin halutaan sisällyttää?

Mallin modulaarisuus nousi esille kolmantena keskeisenä löydöksenä, joka nostettiin haastatteluissa esille devOpsia tukevana arkkitehtuuriratkaisuna. Palvelun paloittelun onnistuminen sekä löysiin liitoksiin perustuva arkkitehtuuri nähtiin parhaiten yhteensopivaksi devOps-toimintamallin kannalta. Myös devOpsin kanssa parina usein esitetty mikropalvelumalli perustuu näihin kahteen piirteeseen. Modulaarisuudesta voidaan nostaa esille seuraavia suunnittelua tukevia kysymyksiä:

1) Miten malli paloitellaan sopiviin kokonaisuuksiin?

2) Miten riippuvuudet hallitaan paloittelun jälkeen?

3) Mahdollistavatko mallin liitokset tuotantoputken automatisoinnin?

Kaikkiin kolmeen löydökseen liittyvät oleellisesti myös taloudelliset ja henkilöstöön liittyvät kysymykset. Organisaation tiimin kyvykkyyksien tulisi soveltua valittuihin

työvälineisiin ja teknologioihin, jotta devOps-ratkaisusta on mahdollista saada toimiva, ja erityisesti onnistuneella muutosjohtamisella voidaan parantaa devOps-toimintamallin sopivuutta organisaatioon. Lisäksi organisaatiossa olisi myös osaamisen tasolla huomioitava devOpsin ja arkkitehtuurin linkittyminen toisiinsa, niin että kommunikaatio devOps-tiimin ja arkkitehtien välillä mahdollistaisi toimivan ratkaisun löytämistä.

Toinen päälöydöksiä rajoittava tekijä on ratkaisun kustannukset, sillä empirian perusteella devOps-ratkaisun supistaminen liittyy usein korkeisiin kustannuksiin. DevOps-kyvykkyyksiä implementoidessa on tärkeää varata riittävä investointi, sillä kustannusoptimoidut ratkaisut ovat empirian perusteella johtaneet toimimattomiin devOps-ratkaisuihin.

6.1 Kriittinen pohdinta ja jatkotutkimus

Tutkimuksen toteutuksessa esiintyi muutamia valintoja, jotka voivat rajoittaa tutkimuksen käytettävyyttä. Ensimmäinen valinta oli tehdä haastattelut ainoastaan yhdessä konsulttiyrityksessä, jolloin otanta esimerkiksi käytetyistä teknologioista voi olla vinoutunut, kun kaikki haastateltavat ovat hyödyntäneet yrityksessä painotettuja teknologioita ja työkaluja projekteissaan. Lisäksi tutkimus rajoittui tutkimaan pääsääntöisesti suuria yrityksiä, sillä sekä empiria että teoria painottuivat suuriin yrityksiin. Suurissa yrityksissä saatavilla olevat resurssit esimerkiksi henkilöstön osaamiseen tai budjettiin liittyen ovat volyymiltään hyvin erilaiset kuin pienissä yrityksissä, jolloin esimerkiksi henkilöstön osaaminen devOps-implementoinnin rajoitteena ei korostunut yhtä vahvasti kuin mitä se voisi korostua, jos yrityksen koko olisi eri.

Yhteenvedossa esitettävää taulukointia voidaan edelleen pitää hyvin suuntaa antavana ohjeena, sillä devOpsin ja arkkitehtuurin mallit vaihtelevat ohjelmistoissa ja organisaatioissa. Sekä devOps että arkkitehtuuri ovat liikkuvia konsepteja, jotka ovat organisaatioissa jatkuvan kehityksen ja muutoksen alla. Tutkimuksesta onkin hyvä huomioida, että empiriasta tehdyt löydökset kuvastavat yhden tietyn projektin nykyhetken tilaa.

Tutkimusta arkkitehtuurien ja devOpsin yhteensopivuuden kannalta voidaan edelleen jatkojalostaa. Baskarada ym. (2018) nostivat tutkimuksessaan esiin mikropalvelumallin soveltumattomuuden monimutkaisempiin järjestelmiin, kuten

toiminnanohjausjärjestelmiin. Saman kaltaisia löydöksiä oli havaittavissa empiriasta, ja esimerkiksi pakettiohjelmistot nostettiin usein esille haastavina devOps-työn kannalta. Jatkotutkimuksena aiheeseen voidaan syventyä vielä tarkemmin pohtimaan, millaiseen arkkitehtuuriin ja ohjelmistoon devOpsin implementointi on ylipäänsä järkevää ja milloin ohjelmiston alkuperäinen rakenne ja käyttötarkoitus ovat liian haastavat devOps-toimintamalliin implementoitavaksi.

7 Yhteenveto

Yhteenvetona tutkimuksen tuloksista muodostettiin taulukointi (taulukko 7) johon nostettiin ylimmälle tasolle tutkimuksen kolme päälöydöstä, jotka parantavat devOps-kyvykkyyksien luomisen onnistumista ohjelmistoarkkitehtuureissa. Päälöydöksien alle nostettiin tutkimuksesta poimittuja havaintoja kysymysten muodossa, joiden tarkoituksena on olla suunnittelun tukena. Alimmalle tasolle nostettiin vielä kaksi rajoitetta, taloudelliset sekä henkilöstöön liittyvät rajoitteet.

Taulukko 7 Tulosten yhteenveto

Onnistumisia	Alustan yhteensopivuus	Työkalujen valitseminen tarpeen mukaan	Mallin modulaarisuus
Suunnittelun tukena	Missä alusta sijaitsee: on-premise, pilvi, jokin muu? Millaisia muokkauksia alustaan on mahdollista tehdä? Mitkä ratkaisut alustassa on suunniteltu valmiiksi devOps-toimintamallia tukevina? Onko alustassa yhteneväisyyksiä, joita voidaan hyödyntää devOps-tuotantoputkessa?	Mitä työkaluja tiimi on käyttänyt aiemmin, millaista osaamista tiimillä on? Mitkä devOps-putken osat tukevat liiketoimintatarpeen täyttämistä? Mitä devOpsia tukevia ratkaisuja arkkitehtuuriin halutaan sisällyttää?	Miten malli paloitellaan sopiviin kokonaisuuksiin? Miten riippuvuudet hallitaan paloittelun jälkeen? Mahdollistavatko mallin liitokset tuotantoputken automatisoinnin?

	Miten konfiguraatioidenhallinta on toteutettu?		
Rajoitteet	Taloudelliset Henkilöstö		

Tutkimuksessa arkkitehtuuria tarkasteltiin sekä mallin että käytettyjen teknologioiden pohjalta, ja tulosten pohjalta voidaan todeta, että parhaiten devOps onnistuu sellaisessa arkkitehtuurissa, jonka alusta tukee olemassa olevien devOps-työkalujen hyödyntämistä, joka on pilkottu sopiviin kokonaisuuksiin ja johon valitaan tarpeeseen sopivat työkalut. Arkkitehtuurin yhteensopivuutta devOps-työkaluihin voidaan tukea tutustumalla ensin alustan ominaisuuksiin ja ohjelmiston rakenteeseen ennen toteutuksen aloitamista. Suunnittelussa voidaan pohtia, miten malli voidaan paloitella kokonaisuuksiin niin, että jatkuva kääntäminen on mahdollista, mutta samalla riippuvuuksien hallinnasta ei muodostu liian monimutkaista. Lisäksi mallissa tulisi kriittisesti pohtia työkalujen sopivuutta ratkaisuun, sekä millainen devOps-malli on ratkaisun kannalta tarpeellista toteuttaa.

Lähteet

Anand, P. – Jarvis, A. – Jose, J. (2022). *Successful Management of Cloud Computing and DevOps*. American Society for Quality Press, Milwaukee.

Astudillo, H. – Marquez, G. – Ponce, F. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*.

Auerbach, C. F. – Silverstein, L. B. (2003). *Qualitative data an introduction to coding and analysis*. New York University Press, New York.

Aurum, A. – Berntsson, C. – Svensson, R (2015). On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, Vol. 57(1), 21–31

Babar, M. – Nasab, A. – Shahin, M. (2021). A Qualitative Study of Architectural Design Issues in DevOps. *Journal of Software: Evolution and Process*

Babar, M. A. – Shahin, M. (2020). On the Role of Software Architecture in DevOps Transformation: An Industrial Case Study. *Proceedings of the International Conference on Software and System Processes*, 175-184.

Babar, A.M. – Liming, Z. – Shahin, M. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, Vol. 5, 3909–394

Balalaie, A. – Heydarnoori, A. – Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. IEEE Software. Vol. 33(3), 42 – 52.

Balalaie, A. – Heydarnoori, A. – Jamshidi, P. – Tamburri, D. A. – Lynn, T. (2018). Microservices migration patterns. *Software: Practice and Experience*, Vol. 48(11), 2019-2042.

Barua, H. 2015. The Role of Configuration Management in a Containerized World. < https://www.infoq.com/news/2015/12/containers-vs-config-mgmt/>, haettu 29.11.2022

Baškarada, S. – Nguyen, V. – Koronios, A. (2018). Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems*.

Bogner, J. – Fritzsch, J. – Wagner, S. – Zimmermann, A. (2019). Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality. *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 187-195.

Bolscher, M – Daneva, M. (2019). Designing Software Architecture to Support Continuous Delivery and DevOps: A Systematic Literature Review. *14th International Conference on Software Technologies (ICSOFT 2019)*

Harvard Business Review, Analytic Services (2019). *Competitive Advantage through DevOps.* < https://hbr.org/sponsored/2019/01/competitive-advantage-through-devOps>, haettu 6.5.2022.

Dubaria, D. – Shah, J. - Widhalm, J. (2018). A Survey of DevOps tools for Networking, *9th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 185-188,

Ebert, C. – Gallardo, G. – Hernantes, J – Serrano, N. (2016). DevOps. *IEEE Software*, Vol. 33(3).

Eriksson, P. – Koistinen, K. (2014). *Monenlainen tapaustutkimus*. Helsinki: Kuluttajatutkimuskeskus.

Gaba, N. S. – Kaur, B. – Kaur, M. – Singh, C. (2019). Comparison of Different CI/CD Tools Integrated with Cloud Platform, *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 7 – 12

Ghantous, G. – Gill, A. (2017). DevOps: Concepts, practices, tools, benefits and challenges. *PACIS2017*.

Jaiswal, M. (2019). Software Architecture and Software Design. *International Research Journal of Engineering and Technology (IRJET) e-ISSN*, 2395-0056.

Joby, P. (2019). Exploring DevOps: Challenges and Benefits. Journal of Information Technology and Digital World. Vol. 01(01), 27 – 37.

Kallio, H. – Pietilä, A-M. – Johnson, M. – Kangasniemi, M. (2016). Systematic methodological review: developing a framework for a qualitative semi-structured interview guide. Journal of Advanced Nursing, Vol. 72(12), 2954 – 2965.

Kousa, J. – Ihantola, P. – Hellas, A. – Luukkainen, M. (2020). Teaching Container-Based DevOps Practices. *Web Engineering*. Cham: Springer International Publishing. 494–502tzke, N. – Quint, P-C. (2017). Understanding Cloud-Native

Applications after 10 years of cloud computing – A systematic mapping study. *Journal of Systems and Software*, Vol 126, 1–16.

Kruchten, P. – Obbink, H. – Stafford, J. (2006). The Past, Present, and Future for Software Architecture. *IEEE software 23.2*. 22–30. Web.

Leite, L. – Rocha, C. – Kon, F. – Milojicic, D. – Meirelles, P. (2019). A Survey of DevOps Concepts and Challenges. *ACM Computing Survey*s. Vol. 6(127)

Liang, P. – Shahin, M. – Waseem, M. (2020). A Systematic Mapping Study on Microservices Architecture in DevOps. *The Journal of Systems and Software*. Vol. 170, 110798–.

Light, B. (2005). Potential pitfalls in packaged software adoption. *Communications of the ACM*, Vol. 48(5), 119-121.

Liu, G. – Huang, B. – Liang, Z. – Qin, M. – Zhou, H. – Li, Z. (2020) Microservices: architecture, container, and challenges. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion*, 629– 635

Lwakatare, T. – Karvonen, T. – Sauvola, T. – Heikkilä, V. – Itkonen, J. – Kuvaja, P. – Mikkonen, T. – Oivo, M. – Lassenius, C. (2019). DevOps in practice: A multiple case study of five companies. *Information and Software Technology*, 114, 217–230.

Lwakatare, L.E. – Kuvaja, P. – Oivo, M. (2016). Relationship of DevOps to Agile, Lean and Continuous Deployment: A Multivocal Literature Review Study. *In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol 10027. Springer International Publishing.

Meghazani, K. (2019). From on-premise erp to cloud erp. In *Advanced methodologies and technologies in business operations and management*, 816-826. IGI Global.

Merriam, B. – Tisdell, E. (2015). *Qualitative Research: A Guide to Design and Implementation*. Jossey Bass, San Francisco.

Mishra, A. – Otaiwi, Z. (2020). DevOps and software quality: A systematic mapping. *Computer Science Review*, Vol. 38, 100308–

Mullans, A. - Shaub, W-P. (2018). DevOps: Any Language, Any Platform with Azure DevOps Projects. *MSDN Magazine Issues*. Vol. 33(5).

Newman, S. (2015) *Building Microservices*. 1st ed. O'Reilly, Beijing.

Pautasso, C. – Wilde, E. (2009). Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. *Proceedings of the 18th International Conference on World Wide Web, WWW 2009.*

Pulkkinen, V. (2013). Continuous deployment of software. *In Proc. of the Seminar*. Vol. 58312107, 46.

Ravichandran, T. – Waterhouse, P. (2016). *DevOps for digital leaders : reignite business with a modern DevOps-enabled software factory*. Apress.

Shahin, M– Ali Babar, M. –Zhu, L. 2016. The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '16)*. ACM. Vol. 44, 1–10.

Sharma, S. (2017). *The DevOps adoption playbook: a guide to adopting devOps in a multi-speed IT enterprise (1st edition*). John Wiley and Sons. Indianapolis

Stolberg, S. (2009). Enabling Agile Testing through Continuous Integration, *2009 Agile Conference*, 369– 374

Stuckey, H. L. (2015). The second step in data analysis: Coding qualitative research data. *Journal of Social Health and Diabetes*, Vol. 3(1), 7–10.

Verona, J. (2016). *Practical DevOps*. Packt Publishing Ltd.

Virmani, M. (2015). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, 78– 82.

Yarlagadda, R. (2018). Understanding DevOps & Bridging the Gap from Continuous Integration to Continuous Delivery. *International Journal of Emerging Technologies and Innovative Research*, Vol. 5(2). 1420– 1424.

Liite A: Haastattelukysymykset

1. Perustiedot

- Oma ura, koulutus
- Kuinka pitkään on työskennellyt devops/arkkitehtuurien parissa?
- Kiinnostuminen arkkitehtuurista/devopsista?

2. Projektit

- Millaisissa projekteissa on työskennellyt (toimiala, ohjelmiston käyttötarkoitus, koko, kompleksisuus)?
- Millaisista ohjelmistoarkkitehtuureista on lähdetty liikkeelle (monoliittinen, SOA-mallinen)?
- Mitä devops työkaluja on otettu käyttöön muutosten yhteydessä (CI/CD-työkalut (Jenkins, GitLab), versionhallintajärjestelmät, konfiguraatioiden hallinnat (puppet, salt stack, ansible)
- Miten devops ja arkkitehtuuri on toimineet yhdessä? Esimerkkejä onnistuneista ja epäonnistuneista tilanteista

3. Haasteet

- Mitkä asiat arkkitehtuurissa ovat aiheuttaneet haasteita devopsin käyttöönotossa? Esimerkkejä?
- Onko projektien onnistumisissa ollut mahdollisesti jotain yhtäläisyyksiä? Jos niin, mitä?

4. Ongelmien ratkaisu

- Miten arkkitehtuurisia ongelmia on ratkaistu?
- Onko joitain asioita jouduttu jättämään ulkopuolelle (esimerkiksi työkaluista), jotka olisivat devops kehityksen kannalta järkeviä mutta arkkitehtuurin kannalta liian työläitä muuttaa? Jos on, niin mitä ja millä perusteella?

Liite B: Datanhallintasuunnitelma

Opiskelijan aineistonhallintasuunnitelma

Tämän dokumentin avulla voit suunnitella tutkimusaineistosi hallintaa. Yksityiskohtaisemmat ohjeet kuhunkin osioon löydät Opiskelijan aineistonhallintaoppaasta.

Tutkimusaineisto

Tutkimusaineistolla tarkoitetaan kaikkea sitä aineistoa, millä tutkimuksen analyysi ja tulokset voidaan todentaa ja toisintaa. Se voi olla esim. erilaisia mittaustuloksia, kyselyistä ja haastatteluista syntyvää dataa, äänitteitä ja videoita, muistiinpanoja, ohjelmistoja, lähdekoodeja, biologisia näytteitä, tekstinäytteitä ja keruuaineistoja.

Listaa alla olevaan taulukkoon kaikki tutkimuksessasi käyttämäsi tutkimusaineisto. Huomaa, että aineisto saattaa koostua useammasta eri aineistotyypistä, muista kirjata kaikki eri aineistotyypit. Listaa sekä digitaalinen että fyysinen tutkimusaineisto.

Aineistotyyppi	**Sisältää henkilötietoja***	**Tuotan aineiston itse**	**Joku muu on tuottanut aineiston**	**Muuta huomioitavaa**
Aineistotyyppi 1: *Haastattelujen nauhoitus*		x		

* Henkilötietoja ovat sellaiset tiedot, joiden perusteella henkilö voidaan tunnistaa suoraan tai välillisesti esimerkiksi yhdistämällä yksittäinen tieto johonkin toiseen tietoon, joka mahdollistaa tunnistamisen. Esimerkkejä henkilötiedoksi katsotuista tiedoista löydät Tietosuojavaltuutetun toimiston sivuilta

Henkilötietojen käsittely tutkimuksessa

Mikäli aineistosi sisältää henkilötietoja, olet velvoitettu noudattamaan EU:n tietosuoja-asetusta (GDPR) sekä Suomen tietosuojalakia. Henkilötietoja sisältävän aineiston osalta sinun tulee laatia tutkittavillesi tietosuojailmoitus sekä selvittää, kuka toimii aineiston osalta rekisterinpitäjänä.

Laadin tutkittavilleni tietosuojailmoituksen** ja toimitan sen heille ennen aineiston keruuta ☐

Henkilötietojen osalta rekisterinpitäjänä** toimii opiskelija ☐ yliopisto ☐

Aineistoni ei sisällä henkilötietoja ☒

**Lisätietoja yliopiston intranetin Tietosuojaohjeita opinnäytetyöhön -sivulta

Aineiston käyttöön liittyvät luvat ja oikeudet

Selvitä mitä lupia ja oikeuksia aineistojen käyttöön liittyy. Ole tarvittaessa yhteydessä opinnäytteesi ohjaajaan. Kuvaile jokaisen aineistotyypin osalta niiden käyttöön liittyvät luvat ja oikeudet, voit tarvittaessa lisätä aineistotyyppejä listaukseen.

Itse tuotettu aineisto

Saatat tarvita erillisiä lupia keräämäsi tai tuottamasi aineiston käyttöön sekä tutkimuksessa että tulosten julkaisemisessa. Mikäli olet arkistoimassa aineistoasi, pyydä tutkittavilta tarvittavat luvat aineiston arkistoimiseen ja jatkokäyttöön. Selvitä myös, vaatiiko valitsemasi arkisto kirjallisia lupia tutkittavilta.

Tarvittavat luvat ja niiden hankkiminen

Aineistotyyppi 1: Haastattelut

- Pyydän haastattelluilta luvan datan käyttämiseen ja tallentamiseen

Jonkun muun tuottama aineisto

Onko sinulla tarvittavat luvat aineiston käyttöön tutkimuksessa ja tulosten julkaisemiseen? Liittyykö aineistoon tekijänoikeuksia tai käyttölisenssejä? Huomioi, että esimerkiksi julkaisujen kuvien ja kaavioiden käyttö saattaa edellyttää lupaa.

Aineiston säilyttäminen tutkimuksen aikana

Missä säilytät aineistoasi tutkimuksen aikana?

Yliopiston verkkokansiossa ☒
Yliopiston tarjoamassa Seafile-pilvipalvelussa ☐
Jossakin muualla, missä? ☐

Yliopiston tallennuspalvelut huolehtivat automaattisesti tietoturvasta ja varmuuskopioinnista. Jos valitset tallentamisen muualle kuin yliopiston palveluihin, kuvaa, miten huolehdit tietoturvasta ja varmuuskopioinnista. Muista varmistaa, mihin tallennat aineiston aina sitä muokattuasi.

Jos käytät tallentamiseen puhelinta, tarkista etukäteen, minne ääni tai video tallentuu. Jos käytät tallentamiseen kaupallisia pilvipalveluita (iCloud, Dropbox, GoogleDrive jne.) ja aineistosi sisältää henkilötietoja, varmista, että tietosuojailmoituksessa antamasi tiedot tietojen siirtymisestä vastaavat laitteistosi asetuksia. Kaupallisten pilvipalveluiden käyttö merkitsee tietojen siirtoa kolmansiin maihin.

Aineiston dokumentointi ja metadata

Miten kuvailet aineistosi niin, että ulkopuolinenkin ymmärtää, millaista aineisto on? Miten itse tarpeen tullen palautat vuosien kuluttua mieleesi, mistä aineistosi koostuu?

Aineiston dokumentointi

Pystytkö kertomaan, mitä aineistollesi on tapahtunut tutkimuksen teon aikana? Aineiston dokumentointi on keskeisessä osassa aineistoon tehtyjen muutosten jäljittämisessä.

Käytän aineiston dokumentointiin

tutkimuspäiväkirjaa ☐
erillistä dokumenttia, johon kirjaan aineiston pääasiat, kuten tehdyt muutokset, analyysin vaiheet sekä esim. muuttujien merkitykset ☒ aineiston mukana kulkevaa readme-tiedostoa, jossa kuvataan aineiston pääasiat ☐ jotain muuta, mitä? ☐

Aineiston järjestys ja eheys

Miten pidät aineistosi järjestyksessä ja ehyenä, ja vältät sen tahattomat muutokset?

Säilytän alkuperäisen aineiston erillään tutkimuksenteon aikana käyttämästäni aineistosta, jotta voin palata alkuperäiseen, jos tarvetta ilmenee. ☒

Versionhallinta: mietin jo ennen tutkimuksenteon alkua, miten tulen nimeämään eri aineistoversiot ja noudan sitä systemaattisesti ☒

Tiedostan jo tutkimuksen alussa aineistoni elinkaaren, ja varaudun tilanteisiin, joissa data saattaa huomaamatta muuttua, kuten esim. nauhoitus, litterointi, konversio toiseen tiedostomuotoon, tallentaminen jne. ☒

Metadata

Metadata on kuvaus aineistostasi. Metadatan perusteella henkilö, joka ei tunne aineistoasi, ymmärtää, millaista aineistosi on. Metadataa voi olla mm. tiedoston nimi, sijainti, koko ja tieto aineiston tuottajasta. Tarvitsetko metadataa?

Tallennan aineistoni arkistoon tai tietopankkiin, joka huolehtii metadatasta puolestani. ☐

Minun pitää luoda metadata, koska arkisto, johon tallennan aineiston edellyttää sitä. ☐

En tallenna aineistoani julkiseen arkistoon, enkä tarvitse metadataa. ☒

Aineisto tutkimuksen valmistuttua

Olet vastuussa aineistostasi myös tutkimuksen valmistumisen jälkeen. Varmista, että käsittelet sitä tekemiesi sopimusten mukaisesti. Yliopiston suosittelema säilytysaika on viisi vuotta, poikkeuksena kuitenkin lääketieteen alan aineistot, joiden säilytysaika on 15 vuotta. Henkilötietoja voi säilyttää vain sen aikaa, kun tarve on. Jos olet sitoutunut tuhoamaan aineiston määräajan päätyttyä, sinun on huolehdittava siitä, vaikka et olisi enää opiskelija. Myös yliopiston tallennusratkaisuja käytettäessä aineiston tuhoaminen on sinun vastuullasi.

Mitä aineistollesi tapahtuu, kun tutkimus valmistuu?

Säilytän kaiken datan kunnes gradu on valmistunut ja hyväksytty.

Jos säilytät dataa, kuvaa, missä:

Yliopiston oneDrive

Aineistonhallintasuunnitelma kannattaa pitää ajan tasalla läpi tutkimuksen.

Lisätietoja Turun yliopiston kirjaston laatimasta Opiskelijan aineistonhallintaoppaasta

Liite C: Datan koodauksen taulukko

Sitaatit	Kategoria 2	Kategoria 1	Avainsanat
”jos joskus ei olekaan niinku valmista tapaa tehdä vaan sillä että sitä devopsia sieltä aika paljon käyttää siihen niin kuin oikeasti aikaa ja energiaa ja rahaa että rakennat.” (H1)	Työkalujen valinnat	Teknologia	Haasteet
”löydät semmoiset niinkun softapaletin mikä toimii” (H1)			
”hyvän ja ja selkeän mallin löytäminen voi olla tosi vaikeata” (H4)			
” historiallisesta syistä se on asetettu niinku tota jollakin skriptillä ja ja ja muulla ja ja tota sitten se että sitä vaikka siihen on nyt asetettu konfiguraatiota klipsuttelemalla sitä järjestelmää eli että me haluttaiskkin deployata se koodi ja me haluttais deloyata ne konfiguraatiot automaattisesti niin semmoiset on niinku vaikeata.” (H1)	Vanhat teknologiset ratkaisut		
”no ehkä silloin ensimmäisessä projekteissa ongelma oli vähän se että devopsissa tää deployment joka on sen ihan ensimmäinen vaihe oikeastaan että tehdään se build tai deployment niin ne sovellukset kun ne on vielä alunperin alunperin asetettu käsin” (H2)			
”sitten niiden riippuvuudet erilaisissa kirjastoissa on aika monimutkaisia” (H2)			

“jos on just tämmöinen niinku yksi iso monoliitti tai jos on tosi pieni tiimi niin se hyöty jää helposti pienemmäksi vaikka se monet monet asiat voi olla nopeampi tehdä tehdä vaan käsin” (H4).			
“paljon on käytössä niin ohjelmistoja edelleen joiden niinku tavallaan ajatusmalli perustuu ehkä niinku semmoiseen vähän vanhempaan ajatteluun ja ja tota niitä on niinku vaikea saada sovitettua tämmöiseen tämmöiseen devopsmalliin” (H5)			
“vähän vanhempaan tekniikkaan perustuvat ratkaisut niin niin voi olla joskus hankala mistä automatisoida buildeja ja deploymentteja”. (H2)			
”että on joudutuu tekee jotain eristettyjä on premise järjestelmiä niin niissä on tavallaan joudutuu keksiin pyörä uudestaan niin nää mä oon nähnyt aina haasteellisina” (H3)	Alustan sopivuus		
”ja siinä on oma oma niinku ohjelmointikielensä niin se on se on aika aika hankalaa” (H4)			
”ongelma on usein siinä että kun paketti softa niin sitä ei olekaan mietitty sillä tavalla että että tämä tulee devopin kautta asetettuna.” (H1)			
”niin sanotaanko että että varsinkin täällä niinku pakettiohjelmistoja niinkun oli ne sitten ehkä laajempia ERP-järjestelmiä tai tai tota tai sitten niin			

kun tämmöisiä suppea suppeampi joku jonkun liiketoiminta alueen valmisohjelmistoja ja siellä on niinku ehkä eniten nähnyt nähnyt sitten tota sellaisia ratkaisuja jotka ei välttämättä niinku devopsmalliin niin hyvin istu" (H5)			
"ne ohjelmistot sinänsä voisi olla niinkun pilvessä ja pilvestä tarjottuja tota niin se ei itsessään ei vielä ole tae siitä se missä se infra on on se sitten pilvessä tai tai tota on-premissä niin se ei sinänsä vielä kerro sitä kokonaiskuvaa siitä että miten hyvin tämä tämmöinen niinku devops toimintamalli istuu sitten sen sen tota ratkaisun ympärillä" (H5)			
"on enemmän siihen ehkä sen muutoksen hallintaa tai sitten niinku sitä organisaation hallitsemista on sitä ja työkalut niinku semmoista kelpaa kelpaa ihmisille." (H1)	Henkilöstö	Organisatoriset	
"mä oon niinku nähnyt sen niinku haasteen siitä että on yritetty käydä käyttää niinku eri tavalla osaavia resursseja joka on johtanut siihen että siitä on tullut kömpelö tai manuaalisesti juttuja sisältävä" (H3)			
"kynnys kynnys niinku muuttaa toimintatapoja on on on iso" (H4)			

”tavallaan ne asiat mitä löydettiin käytännön kautta niinku devopsi pipeline työstä niin ne heijastui siihen arkkitehtityöhön mutta ne ihmiset oli kuitenkin eri ihmisiä” (H3)			
”että ensisijainen syy että joku jätetään ulos on ehkä enemmänkin kustannukset” (H1)	Taloudelliset		
“enemmänkin priorisointi että mitä kannattaa tehdä mistä on mistä on se suurin hyöty” (H4)			
”niiden ongelmat ovat tämmöisiä esimerkiksi että bisnekseen tulee jotain kustannusoptimointi tavoitteita” (H3)			
”ratkaisuja kehitetään tavallaan järkevästi palasteltuina pilkottuna sopiviin sopiviin kokonaisuuksiin” (H5)	Modulaarisuus	Teknologia	Onnistumiset
”on niinku löyhästi riittävän löyhästi toisistaan riippuvia” (H5)			
”sillon monessa ne alkaa uudistaa myös sitä omaa devops pipelinea nyt ainakin tyypillinen tai sitten ollaan että joo päättäneet siirtää vaikka mikroservicearkkitehtuurii n” (H3)			
”no kyl mun mielestä just tämmöiset niinku mikropalvelut pienet palvelut” (H4)			
”on suunniteltu tota ja mä sanoisin tämmöisenä pilvinatiiveina sovelluksina niin tota yleensä niissä se se tota arkkitehtuuri on on sitten sellainen että se	Työkalut		

tukee tukee myös tällaista devopstyyppistä tyyppistä kehittämistä” (H5)			
”nyt ei välttämättä oo voitu käyttää niinku vaikka jotain pilvi ratkaisuja vaan että on joudutuu” (H3)			
“pilvi on ollut että se teknologia joka on mahdollistanut sen että on saanut paljon enemmän oikeuksia ja saanut niinku lupaa tehdä enemmän ja asentaa vaikka automaattisesti sitten tuota tuotantoon asti” (H1)			
”…ja sitten haluttiin käyttää PaaS-palveluita ja azure tietokantapalveluita ja sitten openshift ja konttiteknologiaa.” (H1)			
”no konttiteknologia on sellainen joka joka tota helpottaa devopsin käyttöönottoo” (H2)			
“viimeisen parikolme vuotta niin niin tota onko sitten tota niinku openshift tää konttialustaa ja ja siinä ne ja devopsia mikä ollaan tehty” (H4)			
”kontit on melko samanlaisia kuitenkin eli ne konfiguraatiot ei eroa kauhean paljon” (H2)	Alustan sopivuus		
”kun se on sellainen hyvin yhtenäinen alusta sinänsä niin on huomattu että nopeuttaa paljon” (H2)			
”ne on saman tyyppisiä vielä että että sama malli toimii toimii niille niille kaikille komponenteilla niin semmoisessa niin niistä on paljon hyöty” (H4)			

”Devops varmaan niinkun helpoiten tapahtuu just siinä kun me tehdään jotakin niin kun räätälöity softaa.” (H1)			
”siinä vaiheessa kun laitetaan tavallaan jotain devops kyvykkyyttä niin silloin kannattaa olla osaava tiimi” (H3)	Henkilöstö	Organisatoriset	
”niinku löydettiin tavallaan tällaisia niinku löydettiin tavallaan tällaisia arkkitehtuuriratkaisuja että että käyttämällä tiettyjä niin kun tavallaan pilviagnostisia devops työkaluja ja ja sitten niinku paketoimalla ja ja tota jalkautumalla ne ne tämmöiset kontti konttiteknologian avulla niin tavallaan saatiin aikaiseksi semmoinen niinku alustaratkaisu” (H5)	Pilvi-agnostiset devOps-työkalut	Työkalut	Ratkaisut
”ratkaistiin nimenomaan jos lähdetään näistä ihan ensimmäisistä projektissa missä mä oon ollut niin parannettiin niiden sovellusten paketointia” (H2)	Paketoinnin parantaminen		
”buildia ei tehdä osana sitä devops pipelinea vaan buildi tehdään kehittäjien ympäristössä ja sitten tonne gitti repoon viedään niinku tällainen valmis sovelluspaketti” (H2)	DevOps-putken osien jättäminen ratkaisun ulkopuolelle	DevOpsin muokkaaminen	
”sitten jouduttu jättämään niin tämmöinen automaattisestaus vaikkapa vaikkapa sitten sen devopsautomaatio ulkopuolelle” (H5)			
“näissä nykyisissäkin projektissa on paljon paljon ideoita ja ja meidän			

työlistalla on paljon asioita mutta hyvin hyvin vähän vähän siellä on niinku semmoisia asioita mitä mitä ei voi sitten tehdä eteenpäin ennemminkin se on kyse siitä että sitten priorisoinnista mistä saadaan se paras paras hyöty" (H3)			
"me jouduttiin perumaan päätös ja palaamaan takaisinpäin" (H1)	Ratkaisun hylkääminen		