Mikropalveluarkkitehtuurin haasteet DevOpsissa ja niiden hallitseminen

Turun yliopisto
Tietotekniikan laitos
LuK-tutkielma
Tietojenkäsittelytiede
Huhtikuu 2024
Iiro Partanen

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck-järjestelmällä.

TURUN YLIOPISTO
Tietotekniikan laitos

Iiro Partanen: Mikropalveluarkkitehtuurin haasteet DevOpsissa ja niiden hallitseminen

LuK-tutkielma, 23 s.
Tietojenkäsittelytiede
Huhtikuu 2024

Perinteisen ohjelmistokehityksen suuria monoliittisia sovelluksia on vaikea hallita ja tiheän julkaisutahdin toteuttaminen niillä on hankalaa. Jotta julkaisutahtia voidaan tihentää, tulee yrityksillä olla siihen sopiva arkkitehtuuri. Mikropalvelut ja DevOps ovat nousseet yleiseksi keinoksi lisätä yritysten sisällä ketteryyttä ja mahdollistaa tiheämmän julkaisutahdin. Mikropalveluarkkitehtuuri rikkoo perinteisen ohjelmistokehityksen suuret monoliitit pienemmiksi hallittavimmiksi sovelluksiksi, kun taas DevOps taas antaa viitekehyksen näiden palveluiden hallitsemiselle. Yhdessä ne mahdollistavat tiheämmän julkaisutahdin, sekä tuovat muita hyötyjä. Hyötyjen lisäksi ne myös aiheuttavat omanlaisia haasteitaan.
Tutkielma toteutettiin kirjallisuuskatsauksena ja siinä käsitellään mikropalveluiden ja DevOpsin implementoinnin haasteita ja niiden mitigointia. Tutkielmassa tarkastellaan niin mikropalveluiden, kuin DevOpsin implementointia ja niistä saatuja hyötyjä perinteiseen ohjelmistokehitykseen verrattuna. Tavoitteena on nostaa esiin mikropalveluiden ja DevOpsin yhdessä implementoinnista muodostuneita haasteita ja tämän lisäksi pyritään havaitsemaan keinoja kuinka kyseisiä haasteita voitaisiin mitigoida.
Mikropalveluiden ja DevOpsin implementoinnilla on saavutettu positiivisia tuloksia yrityksissä. Implementointi toi esille haasteita sovelluksen rajapinnoissa, suorituskyvyssä, koonnissa, testaamisessa ja julkaisemisessa, joissa haasteita aiheutti varsinkin palveluiden määrän kasvu ja niiden väliset yhteydet. Näiden lisäksi haasteita muodostui yrityksen sisäisessä rakenteessa ja yleiskustannuksissa. Samoja haasteita esiintyi hyvin ristikkäisissä tutkimuksissa, joissa yritykset olivat eriäviä. Kyseisiä haasteita on pystytty mitigoimaan ja niihin ehdotetaan erilaisia strategioita. Kuitenkaan kaikkia haasteita ei pystytä mitigoimaan ja kyseiset haasteet vaativat vielä jatkotutkimusta DevOpsin ja mikropalveluiden implementoinnista.

Asiasanat: mikropalvelut, DevOps, haasteet, mitigointi

Sisällys

1 Johdanto **1**

2 Mikropalveluiden kehitys DevOps-mallissa **3**

2.1 DevOps toimintamallina 3

2.2 Mikropalveluarkkitehtuuri 7

3 Haasteet mikropalveluiden ja DevOpsin implementoinnissa **10**

3.1 Lisääntyneet sovellukset 10

3.2 Koonti ja testaaminen 12

3.3 Julkaiseminen 15

3.4 Yritysten sisäiset haasteet 17

3.5 Pohdintaa 19

4 Yhteenveto **22**

Lähdeluettelo **24**

1 Johdanto

Teknologian ja asiakkaiden vaatimuksien kehittymisen myötä yrityksillä on jatkuva paine implementoida käyttäjien tahtomia uudistuksia ja päivittää sovelluksiaan entistä nopeammin. Yritykset eivät voi enää päivittää sovellusta vain kerran tai kaksi vuodessa, kerätä palautetta, ja pitää asiakkaita odottamassa seuraavaa julkaisua. Nykyään asiakkaat odottavat yrityksiltä jatkuvaa kanssakäymistä, jolloin he voivat antaa palautetta tauotta ja saada ominaisuuksia implementoitua nopeasti. [1] DevOps on muodostunut toimintamalliksi, joka ratkaisee tämän ongelman mahdollistamalla sovelluksen nopeamman julkaisemisen [2].

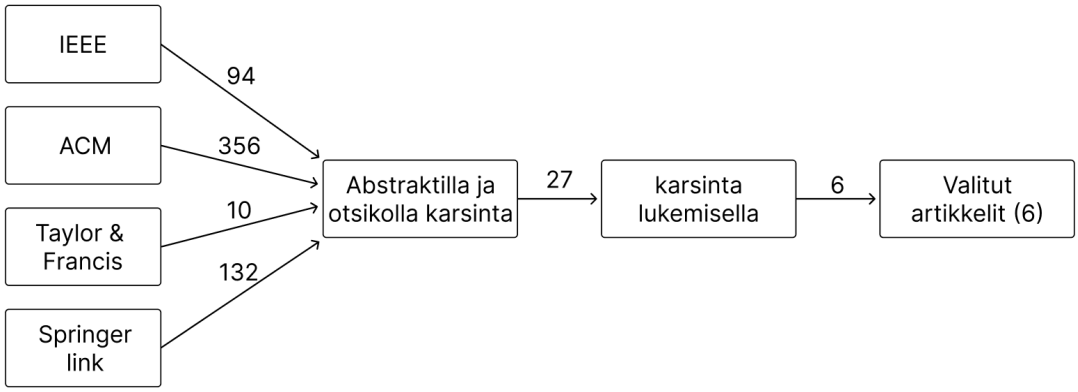
DevOpsin avulla kehityksen syklit voidaan laskea jopa minuutteihin [2], mutta sen implementointi perinteisen ohjelmistokehityksen suurissa sovelluksissa on hyvin vaikeaa. Tämän takia DevOpsin rinnalle on noussut suosioon mikropalveluarkkitehtuuri, joka jakaa suuren sovelluksen pienempiin itsenäisiin sovelluksiin, jotka ovat helpommin hallittavissa ja soveltuvat paremmin DevOpsin kanssa yhteen [3]. Vuonna 2022 47 % yrityksistä hyödynsi DevOps-toimintamallia ja 36 % mikropalveluarkkitehtuuria [4]. Näihin yrityksiin lukeutuu esimerkiksi Amazon, Netflix ja IBM, jotka kaikki hyödyntävät DevOpsia mikropalveluarkkitehtuurin avulla [5].

Mikropalvelut ratkaisevat monoliiteista muodostuvia ongelmia, mutta mikropalveluiden ja DevOpsin yhdistäminen luo omia haasteitaan. Tutkielman on tarkoitus tutkia näitä haasteita ja pyrkiä löytämään keinoja, joilla haasteita voitaisiin mitigoida. Tarkoituksena on vastata kahteen seuraavaan kysymykseen:

TK1: Mitä haasteita mikropalveluiden yhdistäminen DevOpsiin tuo?

TK2: Kuinka näitä haasteita hallitaan?

Tutkielma toteutettiin kirjallisuuskatsauksena. Artikkeleita haettiin eri tietokannoista, joihin lukeutui ACM, IEEE, Taylor & Francis ja Springer Link. Tietokannoista haettiin hakuoperaattorilla ”DevOps AND microservice* AND (challenge* OR issue* OR problem*)”. Löydettyjen artikkelien karsintaa on esitetty kuvassa 1.1. Hakutulokset karsittiin ensiksi abstraktin ja otsikon perusteella, jonka jälkeen jäljelle jääneet artikkelit luettiin kokonaisuudessaan ja valittiin näistä sopivimmat. Valittiin vain artikkeleita, jotka käyttivät englannin tai suomen kieltä ja saavuttivat JUFO-luokituksen 1 tai korkeamman.



Kuva 1.1: Artikkelien karsinta.

Tutkielman toisessa luvussa määritellään tutkielmalle keskeiset käsitteet DevOps ja mikropalvelut, sekä ilmennetään DevOpsin, mikropalveluiden, sekä niiden yhdistämisen mahdollisia hyötyjä ja tavoitteita. Kolmannessa luvussa tehdään kirjallisuuskatsaus mikropalveluiden ja DevOpsin haasteista ja siitä, mitä keinoja on nostettu esille kyseisten haasteiden mitigoimiseen. Tämän lisäksi luvussa on omaa pohdintaa, sekä analyysia aineistoista. Viimeisessä luvussa on yhteenveto tutkielmasta.

2 Mikropalveluiden kehitys DevOps-mallissa

2.1 DevOps toimintamallina

Ohjelmistokehitys on monimutkainen prosessi, joka sisältää monia erilaisia työtehtäviä, kuten kehitystyötä, testausta ja ylläpitoa. Perinteisessä ohjelmistokehityksessä näille tehtäville on omat tehtävien mukaiset tiimit, joiden välille muodostuu usein aukkoja eri työskentelytapojen ja tavoitteiden myötä. [6] DevOps pyrkii kaventamaan näitä aukkoja varsinkin kehityksen ja ylläpidon väliltä lisäämällä ymmärrystä tiimien välillä ja tuomalla kehitys- ja ylläpitotiimien taidot ja kokemukset yhteen [7]. DevOps-toimintamallille ei ole yhtä tiettyä määritelmää, vaan se sisältää erilaisia konsepteja ja sitä kuvaillaan enemmänkin yrityksen sisäisen kulttuurin siirtymisenä yhteistyön, yhteisten tavoitteiden ja luottamisen suuntaan [6].

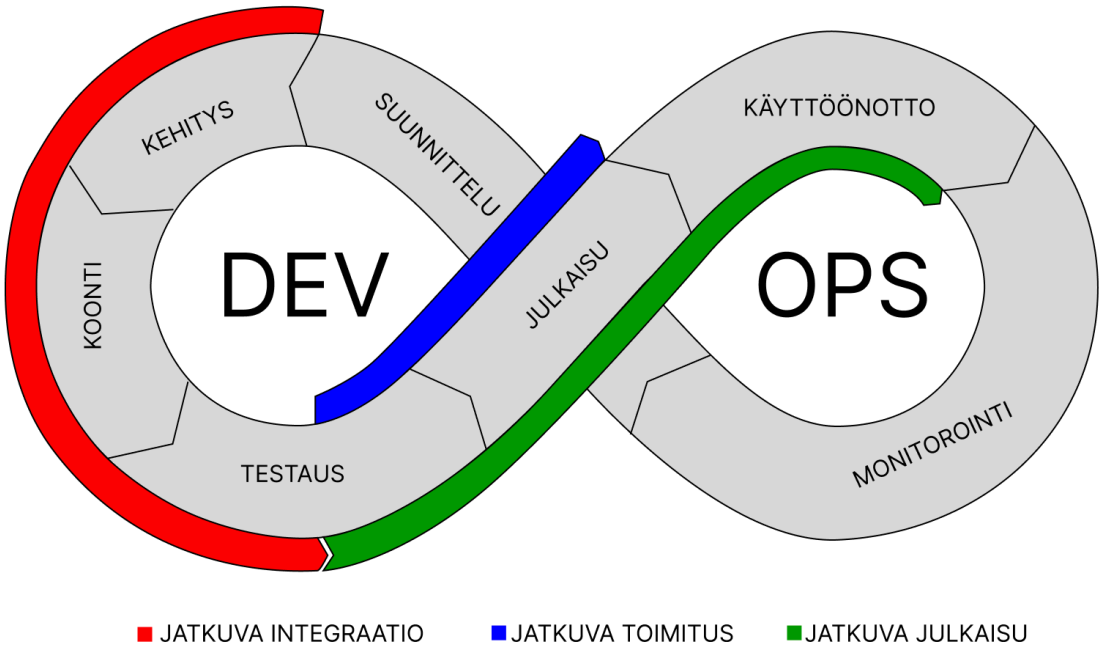
Perinteisessä ohjelmistokehityksessä kehitystiimit pystyvät usein kehittämään toimintoja nopeammin kuin mitä ylläpitotiimit pystyvät julkaisemaan, mikä rajoittaa julkaisuiden määrää [6]. DevOpsissa pyritään ratkaisemaan tätä ongelmaa pitämällä keskeisinä tavoitteina nopeaa julkaisua, jatkuvaa testaamista, nopeampaa reaktiota muutoksiin ja sovelluksen pitämistä julkaistavassa tilassa [1]. Tämän lisäksi keskeistä on kehityksen automatisointi, joka mahdollistaa edellä mainitut tavoitteet. Kehitys- ja ylläpitotiimien välille muodostunutta pullonkaulaa ratkaistaan

tiiviin yhteistyön lisäksi käyttäen niin kutsuttua *jatkuvan integraation* ja *jatkuvan julkaisemisen* prosessia. Jatkuvassa julkaisemisessa nimensä mukaisesti julkaistaan ohjelmisto automaattisesti, kun taas integraatiolla tarkoitetaan prosessia, jossa kaksi sovellusta tai sovelluksen komponenttia integroidaan yhdeksi. Tätä tapahtuu päivittäin monta kertaa ohjelmistokehityksessä, kun kehittäjä integroi omat muutoksensa alkuperäiseen sovellukseen. Sovellusta julkaistaan tuotantoon taas harvemmin ja se vaatii erilaisia testejä ja prosesseja, joten nämä kaksi vaihetta ovat erotettu toisistaan. Jatkuvan integraation ja jatkuvan julkaisemisen avulla integraatiota ja julkaisemista automatisoidaan, jolloin toiminnot pystytään julkaisemaan nopeammin ja mahdollistetaan tiheämpi julkaisutahti [6], [7].

Jatkuva integraatio ja jatkuva julkaiseminen ovat yleisimmin hyödynnetyt DevOpsin vaiheet, mutta DevOpsissa kehitysprosessi muodostuu yhteensä viidestä vaiheesta, jotka ovat *suunnittelu*, *jatkuva integraatio*, *jatkuva toimitus*, *jatkuva julkaisu* ja *monitorointi* [8]. Tämä on esitetty vaiheineen kuvassa 2.1. Tarkoitus on tehdä vaiheista silmukka yhdistämällä viimeinen eli monitorointi ensimmäiseen eli suunnitteluun, jolloin koko ohjelmistokehityksen prosessi pysyy jatkuvana. Koska DevOpsille ei ole tarkkaa määritelmää, ei sille ole myöskään tarkkoja pelisääntöjä, joiden ympärillä toimia. Tällöin DevOpsia voi käyttää implementoimatta jokaista vaihetta tai muuttamalla vaiheita omien tarpeiden mukaisesti.

Jatkuva integraatio ja jatkuva toimitus

Yksi DevOpsin keskeisimmistä vaiheista on jatkuva integraatio, jota hyödynnetään sovellusta kehittäessä. Jatkuvan integraation tavoite on lisätä muutosten jakamista tiimin kesken, sekä koodin jatkuvaa validoimista [1]. Tämä jatkuvuus mahdollistetaan putkistojen (engl. pipeline) avulla, jotka automaattisesti kääntävät, testaavat ja rakentavat ohjelmiston. Putkistoja on erilaisia, mutta yleisesti putkistossa määritellään suoritusympäristö, ympäristössä suoritettavat komennot ja vaiheet,



Kuva 2.1: DevOps-toimintamallin silmukka vaiheineen

jolloin kyseiset komennot suoritetaan. Komennot ovat komentorivikomentoja, joiden avulla voidaan suorittaa suoritusympäristön mukaisia komentoja, esimerkiksi Linux-pohjaisessa sovelluksessa voidaan käyttää Bashia ja sen kautta mitä tahansa Linux-yhteensopivia ohjelmia. Kun kehittäjä tekee muutoksen koodiin ja siirtää sen tiimin tietovarastoon, putkisto käynnistyy automaattisesti [1]. Putkistojen avulla koodia testataan jatkuvasti, jolloin virheet huomataan mahdollisimman aikaisin. Jatkuva toimitus pidentää tätä putkistoa julkaisemalla sovelluksen kvasituotantoon (engl. quasi-production), joka vastaa tuotantoa, mutta on saatavilla vain kehittäjille. Sieltä sovellus voidaan tarvittaessa julkaista nopeasti tuotantoon. [8]

Jatkuva julkaisu

Jatkuvan integraation ja jatkuvan toimituksen avulla sovellusta integroidaan ja julkaistaan tiimin kesken, mutta prosessia voidaan vielä automatisoida tuotantoon julkaisemiseen asti. Yrityksille sovelluksen julkaisemisella on merkittävä vaikutus ohjelmiston viivästymiselle, sillä julkaisun testaaminen manuaalisesti voi perinteisessä

ohjelmistokehityksessä kestää jopa päiviä [1]. Jatkuvassa julkaisemisessa testataan jatkuvan integraation jälkeen vielä sovelluksen toimivuus tuotannossa, luodaan varmuuskopio edellisestä versiosta ja tämän jälkeen viedään julkaisu tuotantoon [8]. Sovelluksen julkistamiseksi tulee sille määritellä suoritusympäristö, johon kuuluu resursseja kuten virtuaalikoneet, verkot ja tietokanta. Jokaiselle julkaistavalle sovellukselle tulee määritellä nämä, joten se muodostuu tärkeäksi varsinkin sovelluksille, jotka koostuvat useammasta pienestä itsenäisesti julkaistusta sovelluksesta. Perinteisesti nämä resurssit konfiguroidaan manuaalisesti, mikä on hidasta. Konfiguraatiota voidaan nopeuttaa määrittelemällä sovellukselle infrastruktuurilähdekoodia, jossa määritellään infrastruktuurin komponentit ja niiden konfiguraatio. Koodi suoritetaan *infrastruktuuri koodina* -työkalun avulla, jolloin sovelluksen infrastruktuurin luominen ja julkaiseminen on mahdollista automaattisesti. [1], [9]

Monitorointi ja suunnittelu

Sovelluksen julkaisemisen jälkeen sovellusta monitoroidaan ja operoidaan. Tämä tapahtuu kuten perinteisessä ohjelmistokehityksessä, mutta DevOpsissa on tärkeää myös sovelluksen toiminnan monitoroinnin lisäksi monitoroida itse DevOpsin toimintaa ja antaa jatkuvaa palautetta eri prosessin vaiheista [8]. Sovelluksen monitoroinnissa seurataan sovelluksen toimintaa, pyritään huomaamaan ongelmia ja analysoidaan sovelluksen suorituskykyä muutoksista. Ongelmien myötä edetään suunnitteluun.

Suunnittelu vastaa myös perinteisestä ohjelmistokehitystä, mutta DevOpsilla mahdollistetaan nopeampi sopeutuminen asiakkaan vaatimuksiin. DevOpsissa pidetään priorisoitua listaa halutuista toiminnoista, jolloin niiden tärkeyttä pystytään päivittämään nopeasti. Tämän lisäksi DevOpsissa kehityksen syklit tulee pitää lyhyinä suunnittelemalla vain pieniä osia sovelluksesta, toteuttamalla ne, saamalla palautetta ja reagoimalla tähän palautteeseen nopeasti seuraavassa syklissä. [1]

2.2 Mikropalveluarkkitehtuuri

Perinteisessä ohjelmistokehityksessä sekä sovelluksia luodessa hyödynnetään usein monoliittista arkkitehtuuria. Monoliiteissa koko sovellus toimii yhdessä ympäristössä, joka sisältää kaiken tarvittavan, kuten käyttöliittymän, palvelinpuolen logiikan ja tietokannan yhdessä. Yksittäinen kehittäjä työskentelee aina koko sovelluksen parissa, ja sovellusta päivittäessä tulee koko sovelluksen edellinen versio poistaa käytöstä ja korvata uudella versiolla. Monoliittisia sovelluksia käsitellään aina vain yhdessä suoritusympäristössä, jolloin ne ovat helpommin testattavissa, julkaistavissa ja monitoroitavissa kuin suuret mikropalveluihin perustuvat sovellukset, jotka koostuvat eriävistä suoritusympäristöistä. Tämän lisäksi kaikki data monoliitissa on yhden sovelluksen sisällä ja sovelluksen ei tarvitse lainkaan kommunikoida sen ulkopuolelle. Monoliittisia sovelluksia suositaan niiden yksinkertaisuuden takia varsinkin sovelluksen elinkaaren alussa. [10] Kuitenkin sovelluksen kasvaessa alkaa muodostumaan ongelmia. Suurien monoliittien muokkaaminen alkaa vaikeutua, koska kun yhtä osaa monoliitista muokataan, saattaa se tahtomatta vaikuttaa sovelluksen toiseen osaan. Monimutkaisuus voi vaikeuttaa myös muutosten pitämistä vain muokatun osan sisällä ja muutokset propagoituvat sovelluksen muihin osiin. Tällöin ei-toivotut muutokset tulee korjata sovelluksen muista osista, mikä lisää kehitysaikaa ja uusien ominaisuuksien lisäämisestä tulee riskialtista. [11]

Mikropalveluarkkitehtuuri taas jakaa monoliittisen sovelluksen pienempiin modulaarisempiin palveluihin, joita voidaan kehittää, testata ja julkaista itsenäisesti [11]. Yksittäinen sovellus voi sisältää kymmeniä tai jopa satoja palveluita [12], jotka kommunikoivat toistensa kanssa kevyillä protokollilla, kuten HTTP:llä. Mikropalveluille on kolme tyypillistä periaatetta. Ensimmäisenä on autonomisuus eli mahdollisuus julkaista yksittäinen palvelu itsenäisesti ilman muita sovelluksen osia, jolloin yhtä mikropalvelua voidaan hyödyntää alkuperäisen sovelluksen ulkopuolellakin. Toisena on palvelun pitäminen pienenä, jolloin yksittäisellä palvelulla tulisi

olla vain yksi vastuualue eikä koskaan tulisi kahta palvelua, jotka vastaavat samasta vastuualueesta. Viimeisenä on palveluiden pitäminen ensiluokkaisina (engl. first-class citizen), jolloin palvelu paljastaa vain julkisen rajapinnan eikä sen tule paljastaa lainkaan sen logiikkaa, arkkitehtuuria tai hyödynnettyjä teknologioita. [10] Mikropalveluiden kehittäminen saattaa viedä enemmän aikaa, koska jokaista palvelua kehitetään ja testataan itsenäisesti, mutta sovelluksen kasvaessa monen pienen palvelun kehittäminen voi olla helpompaa kuin yksittäisen suuren monoliitin [11].

Mikropalveluarkkitehtuuri mahdollistaa myös uusia hyötyjä monoliitteihin verrattuna. Suurimpien hyötyjen väitetään tulevan yksittäisten palveluiden välille muodostuneesta kommunikoinnista, jolloin eri palvelut voivat käyttää eri ohjelmointikieliä ja teknologioita [11]. Tämän lisäksi palveluiden välinen löyhä kytkös (engl. loose coupling) rajoittaa virheiden leviämistä yksittäisen palvelun sisällä, jolloin yksittäisen palvelun kaatuessa sovellus pysyy edelleen toiminnassa ja vain palvelun vastaama osa-alue lakkaa toiminnasta [10].

Monoliitin jakaminen pienempiin sovelluksiin voi mahdollistaa myös helpomman skaalaamisen, kun koko sovelluksen skaalaamisen sijasta voidaan skaalata vain pullonkaulana toimiva palvelu. Jotta mikropalveluiden skaalausta voitaisiin hyödyntää, tarvitsevat ne julkaisuun teknologian, joka on kevyt, skaalattava ja helposti siirrettävä. Tähän usein hyödynnetään kontittamista. Kontittamisessa yksittäinen sovellus julkaistaan konttiin (engl. container), joka sisältää sovelluksen suoritusympäristön. Toisin kuin virtuaalikoneet, joissa sovelluksia voidaan myös julkaista, kontit eivät sisällä käyttöjärjestelmän ydintä (engl. kernel), vaan pelkästään käyttäjätilan sovelluksia ja niiden riippuvaisuuksia. Tämä vähentää tarvittavia resursseja ja täten nopeuttaa sovelluksen käynnistämistä ja helpottaa ylläpitoa. [3]

Yksittäisten tiimien pienen koon ja itsenäisyyden komponenttien kehityksessä väitetään mahdollistavan nopeamman iteraation ja kehityksen. Palveluiden pitäminen pienenä vaikuttaa helpottavan myös automaatiota ja täten mahdollistaa Dev-

Opsin jatkuvan integraation ja jatkuvan julkaisemisen, mikä on lähes mahdotonta toteuttaa suurissa monoliiteissa [3]. DevOps antaa kehykset mikropalveluiden nopealle julkaisemiselle ja integraatiolle, kun taas mikropalvelut mahdollistavat tehokkaamman kehityksen [11]. Mikropalveluiden hyödyistä huolimatta arkkitehtuurilla on omat ongelmat ja kun DevOpsia yritetään hyödyntää mikropalveluiden yhteydessä, muodostuu lisäksi uusia haasteita [10].

3 Haasteet mikropalveluiden ja DevOpsin implementoinnissa

3.1 Lisääntyneet sovellukset

Mikropalveluiden implementointi muuttaa sovelluksen rakennetta siten, että sovellus jaetaan pienempiin palveluihin, mikä lisää niin sovellusten määrää kuin myös yhteyksiä sovellusten välillä. Tämä muutos voi ajoittain olla positiivista mahdollistaen luvussa 2.2 mainittuja ominaisuuksia, mutta luo myös haasteita mikropalveluiden implementaatioon ja DevOpsiin. Palveluiden määrän kasvu vaikuttaa sovelluksen monimutkaisuuteen ja lisää sovellusten välisiä riippuvuuksia [11], mikä vaikeuttaa sovelluksen ylläpitoa ja vaatii yritykseltä tiivistä yhteistyötä ja laadukkaita ohjesääntöjä sovellusten ylläpitoon. DevOpsin ja mikropalveluiden yhdistämiselle ei kuitenkaan ole yleisiä ohjesääntöjä, joiden avulla ne voitaisiin implementoida. Ohjesääntöjen puuttuminen voi aiheuttaa ohjelmistokehittäjissä hämmentymistä ja johtaa jopa mikropalveluiden sulautumiseen takaisin yksittäiseksi monoliitiksi [13]. Jotta tältä pystyttäisiin välttymään, tulisi yrityksen sisällä luoda mikropalveluiden ja DevOpsin käytölle omat ohjesäännöt, joita noudatetaan.

Lisääntyneet yhteydet palveluiden välille vaativat yrityksiltä huolellista rakennetta rajapintojen suunnitteluun. Huolimaton rajapintojen suunnittelu voi johtaa rajapintoihin, jotka ovat virheherkkiä ja vaikeita käyttää. Jotta tätä voidaan mi-

tigoida, tulee yrityksen lisätä ohjesääntöihin tiimien väliset säännöt rajapintojen määrittelyyn. Chen [14] ehdottaa yrityksille kahta seuraavaa strategiaa: ensinnäkin palveluiden välisten yhteyksien pitäminen yksinkertaisena, jolloin jokaista lisättävää yhteyttä tulee kriittisesti tarkistella ja varmistaa, että implementoidaan vain tarvittava sisältö. Toiseksi siirrettävässä datassa tulee seurata myös yksinkertaisuutta eli palveluiden tulee vastaanottaa dataa mahdollisimman laajasti ja lähettää vain tarpeellinen. Näiden avulla palveluiden väliset suhteet pystytään pitämään mahdollisimman löyhinä ja mahdollistetaan tarvittaessa pieniä muutoksia rajapintoihin rikkomatta palveluiden välistä yhteyttä. [14]

Mikropalveluiden väliset uudet yhteydet vaikuttavat myös sovelluksen suorituskykyyn. Lisääntyneet palvelut aiheuttavat enemmän ulkoisia kutsuja, jotka heikentävät sovelluksen suorituskykyä ja altistavat useammille virheille [11], [15]. Yritysten tulee implementoida vakaa verkkoinfrastruktuuri ja painottaa ohjelmistokehityksen monitorointivaihetta, jotta voidaan taata vankka yhteys ja mitigoida haasteet nopeasti [11]. Vakaan infrastruktuurin implementointi parantaa sovelluksen suorituskykyä, mutta ei takaa virheettömyyttä ja täydellistä suorituskykyä. Tulee siis miettiä sovellusta suunnitellessa, ovatko mikropalvelut sopivia sovelluksille, jotka vaativat vakaata toimintakelpoisuutta tai erittäin pientä viivettä. Aina ei ole kannattavaa implementoida mikropalveluarkkitehtuuria, joten kehityksen suunnitteluvaiheessa tulee pitää tarkasti mielessä sovelluksen suorituskykyvaatimukset [13].

Monoliitin jako mikropalveluihin mahdollistaa myös eri teknologioiden hyödyntämisen palveluissa, jolloin jokainen sovellus voidaan kehittää juuri sille sopivalla teknologialla. Eri teknologioiden käyttäminen täysin vapaasti voi kuitenkin johtaa aukkojen syntymiseen tiimien välille. DevOpsille tärkeä yhteistyö vaikeutuu, jos tiimit hyödyntävät täysin eri teknologioita, joista toinen ei tiedä mitään. Tällöin esimerkiksi kehittäjien siirtäminen palvelusta toiseen vaikeutuu huomattavasti, kun kehittäjälle tulee opettaa sovelluksen teknologiat uudestaan. Jonka lisäksi jokainen

uusi teknologia, jota tuodaan yritykseen, lisää yrityksen yleiskustannuksia. [14] Tämän lisäksi uudet teknologiat voivat myös aiheuttaa haasteita integraation kanssa jo implementoituihin teknologioihin, mikä vaikeuttaa sovellusten välisestä kommunikaatiota [11]. Tätä voidaan mitigoida yrityksen sisäisen yhteistyön ja suunnittelun avulla, jolloin voidaan hyödyntää prosessia, jonka jokainen uusi haluttu teknologia joutuu läpäisemään. [14] Tässä prosessissa mietitään teknologian hyödyt, haitat ja sen sopivuus yrityksen sisäiseen toimintaan, jolloin pystytään hallitsemaan yrityksen sisäisten teknologioiden määrää ja valittuja teknologioita.

Jokainen sovellus mitä yrityksen sisällä luodaan, lisää monimutkaisuutta, mutta myös yleiskustannuksia, varsinkin silloin kun yksittäistä sovellusta joudutaan julkaisemaan ja monitoroimaan itsenäisesti [11]. Sovelluksen kompleksisuuden lisäksi mikropalvelun lisääminen kasvattaa organisaation kokoa, kun uuteen sovellukseen tarvitaan uusia työntekijöitä. Yritykset voivat myös päätyä väärinkäyttämään mikropalveluita luomalla niitä liiaksi [13]. Jotta mikropalveluiden liiallista käyttöä pystytään hallitsemaan DevOpsissa ja ohjelmistokehityksessä, tulee tärkeäksi juuri yrityksen sisäiset ohjesäännöt. Tämän lisäksi kehittäjien ja yritysten tulee tarkasti suunnitella mikropalveluarkkitehtuurin sopivuutta suunniteltuihin muutoksiin, koska uusi mikropalvelu ei sovi jokaiseen tapaukseen [11], [13]. Mikropalveluita hyödyntäessä tulee olla tarkkana, ettei niiden käyttö ole holtitonta, vaan palveluita käytetään optimaalisesti. Mikropalveluiden implementointi on haastavaa ja vaatii enemmän osaamista kuin monoliittisen sovelluksen toteuttaminen, jolloin tätä yleiskustannuksen nousua ei voida mitigoida.

3.2 Koonti ja testaaminen

Jatkuvan integraation ja jatkuvan julkaisemisen keskeisiin vaiheisiin kuuluu sovelluksen testaaminen. Mikropalveluiden implementointi tuo sovelluksen testaamiseen monimutkaisuutta luomalla suuren määrän erilaisia palveluita, joista jokaisella on

oma roolinsa sovelluskokonaisuudessa. Tällöin jokaista palvelua tulee testata yksittäisenä palveluna, mutta myös osana suurempaa systeemiä, jolloin taataan itse palvelun toimivuus sekä koko sovelluksen toiminta. Testaamisen tärkeys ja monimutkaisuus kasvaa monoliitteihin verrattuna. [11]

Mikropalveluista muodostuneet uudet palvelut vaikuttavat edellisessä luvussa mainittujen asioiden lisäksi testaamiseen, jolloin uusia haasteita luo palveluiden väliset yhteydet [14]. Monoliitteihin verrattuna mikropalveluissa täytyy testata yksittäisen palvelun lisäksi palveluiden välisiä yhteyksiä, joiden testaamiseen voi kuulua integraatiotestausta, systeemitestausta ja hyväksymistestausta [16]. Integraatiotestauksessa testataan sovelluksen välisten palveluiden toimintaa ja pyritään löytämään virheitä palveluiden välisestä kommunikaatiosta. Systeemitestauksessa taas testataan sovellusta kokonaisuudessaan, jolloin voidaan testata, että sovellus vastaa haluttuja tavoitteita. Lopuksi vielä hyväksymistestauksessa testataan sovelluksen valmiutta julkaisuun. Näiden lisäksi yksittäistä sovellusta voidaan testata vielä yksikkötestien avulla. Testaaminen vaatii lisää osaamista ja on vaikeampaa, kun sovellukselle lisätään testattavia osia, joten mikropalveluiden testaamisen väitetään olevan vaikeampaa kuin keskitetyiden monoliittien, joissa sovellusta testaan vain yhtenä suuren yksikkönä. [16]

Testaamisen määrän kasvamisen lisäksi palvelut heikentävät sovelluksen vakautta, koska lisääntyneet palvelut ja palveluiden väliset kommunikaatiot luovat uusia mahdollisia häiriökohtia. Suuren mikropalveluarkkitehtuuriin perustuvan sovelluksen kaikkia mahdollisia virhekohtia on lähes mahdotonta ennustaa. Tämän lisäksi myös lisääntyneet palvelut ja teknologiat luovat uutta konfiguroitavaa sovellukselle, jolloin palveluiden välisissä testeissä joudutaan ajoittain pystyttämään paljon ympäristöjä, jotka ovat hitaita pystyttää, mikä voi huomattavasti hidastaa suurien mikropalveluiden testaamista. [15]

Testaamisen haasteiden mitigoimiseen Chen [14] väittää auttavan neljä seu-

raavaa strategiaa: ensimmäiseksi mikropalveluiden testauksen tärkeyttä pystytään hallitsemaan pitämällä testausta mukana suunnittelussa, jolloin jo suunnittelussa päätetään kehittäjien ja testaajien välillä vaaditut ominaisuudet, ja niiden testaus suunnitellulle muutokselle. Tämän lisäksi testauksen koodia tarkistetaan yhtä huolellisesti kuin itse sovelluksen koodia ja putkistot on asetettu epäonnistumaan, jos sovelluksen testit eivät kata yli 90 % koodista. Näiden avulla pystytään takaamaan tarpeeksi hyvä testaus sovelluksille ja siirtämään testausta syklin alkuun, jolloin kehittäjien tulee jo suunnittelussa miettiä sovelluksen testausta. Toisena tulee sopia asiakkaiden kanssa testeistä, joilla määritellään tahdotut ominaisuudet tarkasti. Implementoimalla nämä testit putkistoon, pystytään takaamaan, etteivät sovelluskehityksen aikaiset muutokset vaikuta asiakkaan vaatimuksiin. Kolmantena on sovellukselle aina pystyssä olevan testiympäristön luominen, jolloin kauan kestävää ympäristön pystyttämistä voidaan välttää. Tällöin putkisto automaattisesti testaa palvelun tässä valmiiksi määritellyssä ympäristössä, jolloin voidaan taata palvelun toiminta toisten palveluiden kanssa ja välttää pystytykseen kuluva aika. Viimeisenä on vielä testaaminen tuotannossa, jossa parannetaan lokien keräämistä (engl. logging) ja monitorointia siten, että tuotannossa ilmenneisiin virheisiin ja häiriöihin voidaan reagoida nopeammin. [14]

Testaamisen lisäksi haasteita syntyy putkistoissa ohjelmakoodin kokoamisen yhteydessä. Uusien palveluiden ja teknologioiden myötä palvelun riippuvuuksien (engl. dependency) määrä kasvaa, mikä vaikeuttaa sovelluksen konfiguraatiota ja kokoamista [17]. Perinteisesti julkaistuja sovelluksia jaetaan virtuaalikoneissa, jossa pyörii aina yksi sovellus ympäristöä kohden. Uusien konfiguraatioiden myötä päivittäminen muodostuu vaikeaksi, sillä kun sovellukseen tuodaan uusi riippuvuus, tulisi yksitellen jokaisen palvelun oma kone päivittää uusilla riippuvuuksilla manuaalisesti [17]. Tämä pystytään mitigoimaan käyttämällä kontittamista. Docker on yksi mahdollinen sovellus kontittamisen hyödyntämiseen. Docker pystyy rakentamaan niin sanottuja

kuvia (engl. image) lukemalla tiedostosta komentorivikäskyjä, joiden avulla määritellään haluttu kuva, joka määrittelee kontin ympäristön. Tämän tiedoston avulla vaaditut riippuvuudet pystytään määrittelemään vain kerran yhdessä tiedostossa ja kaikki tiedostoa käyttävät palvelut pystyvät päivittymään samanaikaisesti. [17]

3.3 Julkaiseminen

Vaikka yrityksen sisällä noudatettaisiin laadukkaita ohjesääntöjä ja palvelut olisivat luotu mahdollisimman joustaviksi, joudutaan silti ajoittain päivittämään palveluiden rajapintoja laajemmin ja siten, että palveluiden välinen yhteensopivuus katkeaa [14], [15]. Perinteisen ohjelmistokehityksen päivityksessä sovelluksen edellinen versio poistettiin toiminnasta ja päivitettiin uudella. Jos tätä hyödynnetään mikropalveluiden päivittämisessä, se johtaisi rajapinnan kanssa keskustelevien palveluiden yhteyden katkeamiseen, kunnes kyseiset palvelut päivitetään. Palveluiden päivittämiseen voi kulua päiviä, mikä katkaisee sovelluksen toiminnan sietämättömiksi ajoiksi. Tällöin kaikki vaikutuksissa olevat palvelut tulisi päivittää samanaikaisesti, mikä vaatisi paljon koordinaatiota tiimien välillä ja olisi yleisesti vaikea toteuttaa. Jotta palveluiden yhteyden välisiltä rikkoutumisilta vältytään, tulee luoda edellisen rajapinnan rinnalle uusi päivitetty rajapinta ja antaa rajapinnan kanssa keskusteleville palveluille aikaa siirtyä käyttämään uutta rajapintaa. Täten sovelluksen palveluita pystyy päivittämään sen ollessa käynnissä. [14] Kyseisen migraation ongelma on taas se, kuinka voidaan varmistaa, että kaikki palvelun kanssa kommunikoivat palvelut ovat päivittyneet uuteen rajapintaan [14], [16].

Palveluiden päivityskin vaatii yrityksen yhteistyötä. Tiimien välille tulee luoda yhtenevät säännöt siitä, miten ja milloin tulee olla siirrytty uusiin palveluihin ja milloin vanhat palvelut voidaan ottaa pois käytöstä. Vaihtoehtoja on monenlaisia ja Chen [14] suosittelee laatimaan tietyn aikarajan, jonka jälkeen vanhan rajapinnan toiminta katkaistaan. Tämän jälkeen vastuu siirtymisestä on rajapinnan kanssa

kommunikoivilla palveluilla. Ajaksi voidaan määritellä esimerkiksi kaksi viikkoa, jonka jälkeen tulee olla siirrytty uuteen rajapintaan [16]. Rajapinnan katkaiseminen on hyvin karu tapa leikata palveluita katki, joten Chen suosittelee rajapinnan käyttäjien muistuttamista siirrosta. Muistuttaminen voidaan tehdä monitorointivaiheessa rajapinnan ilmoittamalla sen olevan siirtymässä poistettavaksi tai ajoittain rajapinnan poistaminen hetkeksi, jolloin rajapintaa käyttävät palvelut huomaavat käyttävän poistettavaa palvelua ja heille voidaan ilmoittaa siirtymisestä. Jos rajapinnan poistamisella ei ole kiirettä, voidaan myös laadukkaalla monitoroinnilla seurata monitorointivaiheessa rajapinnan toimintaa ja poistaa se vasta kun rajapintaa ei käytetä enää lainkaan. [14]

Migraatioaika palveluiden päivittämiseen sopii hyvin DevOpsiin, mutta tällöin tulee olla huolellinen DevOpsin syklien kestossa. Liian pitkät syklit pakottavat päivitykseen kesken syklin, mikä vaikeuttaa päivitysten suunnittelua, kun taas lyhyellä syklillä uuteen rajapintaan päivittäminen voidaan ottaa seuraavaan sykliin mukaan ja suunnitella huolellisesti. Migraatioajan lisäksi DevOps mahdollistaa sovelluksen tiheän päivitystahdin, jolloin päivitettyyn rajapintaan siirtyneiden palveluiden hidasteeksi muodostuu päivityksen vaativien muutosten identifiointi. Callanan ja Spillane [16] suosittelevat vastuuta päivittävälle palvelulle, jossa heidän tulee ilmoittaa, kuinka uuteen rajapintaan pystytään siirtymään. Tällöin päivittäminen uuteen rajapintaan helpottuu.

Itse päivittämisessä syntyy samoja haasteita kuin kappaleessa 3.2 mainituissa jatkuvan integraation ongelmissa. Yksittäisen palvelun päivittäminen on todella hidasta, jos julkaisemisessa hyödynnetään taas yhtä virtuaalikonetta per palvelu. Tällöin palvelut päivitetään yksitellen, mikä on hidasta, koska yksittäisen virtuaalikoneen pystytykseen kuluu huomattava aika. Sovelluksen koon kasvaessa tämä hidastaa putkistoja niin, että niistä saadut hyödyt eivät ole enää kannattavia. Kuten jatkuvassa integraatiossa, myös tässä kontittaminen on ratkaisu sovelluksen no-

peampaan päivittämiseen ja mahdollistaa nopeamman jatkuvan julkaisemisen putkiston. [17] Päivittämisen yhteydessä haasteita saattaa ilmentyä myös palveluista koostuvan rakenteen muuttuessa.

Monoliiteissa on yleisesti yksi suuri tietokanta, jonka tyyppi on ennalta määritelty ja koko sovelluksen tulee käyttää sitä. Mikropalveluissa taas jokainen palvelu voi käyttää itselleen parhaiten sopivaa tietokantaa, mikä mahdollistaa datatyppien optimoimisen ja helpottaa tietokannan kanssa toimimista sen koon pienenemisen myötä [14]. Haasteita kuitenkin muodostuu, kun mikropalveluiden tietokantoja tai tietokantojen sisäistä dataa tulee päivittää. Tietokantaa päivittäessä tulisi päivittää sen sisältävä data uuteen, mutta mikropalveluissa jokaisella palvelulla on oma tietokanta, jolloin sama data voi kuulua moneen täysin erilliseen tietokantaan. Tällöin tietokantojen välillä ei ole lainkaan linkkiä toisiinsa, joten päivittämistä ei voida ketjuttaa tietokannasta toiseen, mikä johtaa datan epäjohdonmukaisuuteen ja ongelmiin sovelluksessa. Sama pätee myös itse tietokannan päivittämiseen, jolloin päivitys voi aiheuttaa datan tyyppieroja palveluiden välillä, jos vain toisen palvelun data päivitetään. [11], [15], [16] Baškarada, Nguyen ja Koronios [15] ehdottavat yksittäisen suuren tietokannan pitämistä, mikä mitätöisi kaikki mikropalveluista saadut hyödyt tietokantojen suhteen. Toisena Callanan ja Spillane [16] ehdottavat datan muutosten pitämistä taaksepäin yhteensopivana, mikä taas vaikeutti muutosten tekemistä huomattavasti ja kehittäjät tyytyivät tekemään päivityksiä käsin, mikä mitätöi jatkuvan julkaisun automatisoinnin.

3.4 Yritysten sisäiset haasteet

DevOpsin ja mikropalveluiden implementoinnin haasteet eivät ylety vain tekniselle tasolle vaan varsinkin DevOps luo yrityksen sisäisiä haasteita, joihin tulee reagoida. Monoliittisessa sovelluksessa yksi kehittäjä voi erikoistua yhteen teknologiaan tai ominaisuuteen riippumatta muun sovelluksen rakenteesta. Mikropalveluissa taas

palvelut koostuvat pienistä tiimeistä, joissa yksittäiseltä kehittäjältä vaaditaan osaamista laajemmin koko palvelun toiminnasta, jonka lisäksi DevOps tuo kehittäjälle lisää opeteltavaa ohjelmistokehityksen sykleistä. [15] Tämä luo haasteita niin uusien kehittäjien rekrytointiin kuin myös mikropalveluarkkitehtuuriin siirtymisessä. Yritysten täytyy tarkkaan suunnitella mikropalveluarkkitehtuurin sopivuutta omien kehittäjien taitojen mukaan. Kehittäjiä rekrytoidessa on hyvä myös huomioida vaadittu laaja osaaminen palveluiden toiminnasta. Kehittäjien teknologisten taitojen lisäksi kannattaa opettaa kehittäjille DevOpsin ja mikropalvelun implementoinnista ja siitä kuinka ne voivat vaikuttaa toisiinsa. Tämä voi auttaa kehittäjiä ymmärtämään paremmin ohjelmistokehitystä ja näin tukien työskentelyä tiimeissä [13].

Kehittäjien lisäksi DevOpsiin siirtyminen tulee toteuttaa myös johtajille. Johtajien vähäinen ymmärrys DevOpsista vaikuttaa kehittäjien työskentelyyn, sekä toisiin tiimin jäseniin. Jotta koko yrityksen välistä yhteistyötä ja kommunikaatiota voi toteuttaa, tulee myös ylempää johtoa kouluttaa DevOpsin perusteisiin. Yrityksessä pystytään saavuttamaan DevOpsin ja mikropalveluiden hyödyt vain, jos yritys kokonaisuudessaan sitoutuu noudattamaan ohjesääntöjä ja luottaa DevOps-toimintamalliin. [13] Kummatkin näistä haasteista pystytään ratkaisemaan vain investoimalla lisää työntekijöiden koulutukseen. Yritykselle kokeneemmat työntekijät, lisääntyneet palvelut ja panostus eri ohjelmistokehityksen vaiheisiin nostattaa mikropalveluihin ja DevOpsiin siirtymisen hintaa [11], [13], [14], [16], [17]. Hintaa voidaan alentaa luomalla leikkauksia yrityksen sisäisissä ohjesäännöissä, mutta tämä tulee vaikuttamaan DevOpsista ja mikropalveluista saatuihin hyötyihin eikä mahdollista täydellistä implementointia. Hinnan lisäksi DevOpsin ja mikropalveluiden implementointi ei ole nopeaa ja siihen kuuluu hyvin paljon aikaa. Erilaisten teknologioiden opiskeluun ja putkistojen luomiseen voi kulua jopa vuosia [16]. Kummatkin näistä haasteista on mitigoitavissa vain investoimalla toteutukseen.

DevOps ja mikropalvelut vaativat investoinnin lisäksi yrityksen sisäiseltä raken-

teelta muutoksia. Perinteisessä ohjelmistokehityksessä yrityksessä voi olla suuretkin muurit osastojen välillä ja pienetkin muutokset voivat vaatia hyväksynnän useilta ihmisiltä [14]. DevOpsin nopeassa ohjelmistokehityksessä tämä ei ole mahdollista, vaan päätökset muutoksista tulee tehdä DevOpsia harjoittavan pienen tiimin sisällä. DevOpsissa tulee lisätä yrityksen sisäistä luottamusta ja vähentää kehittäjien ja johtajien välistä byrokratiaa. [13], [15] Monoliittisesta arkkitehtuurista siirtyessä yrityksen rakennetta tulee tutkia tarkasti ja se voi vaatia suuriakin muutoksia siinä, kuka vastaa mistäkin palvelusta. Yrityksen rakenne tulee toteuttaa siten, että osaaminen jakautuu tasaisesti palveluille. Yrityksen tulee myös luottaa mikropalveluista ja DevOpsista saataviin hyötyihin ja implementoida niitä, koska mikropalveluiden hyödyt lisääntyvät sen myötä, mitä useampi palvelu yrityksestä löytyy. [15]

3.5 Pohdintaa

Aineistoissa esiteltiin laadukkaasti eri yrityksissä ilmeneviä haasteita ja kuinka niitä on pystytty mitigoimaan. Aineistoihin pohjautuvien yritysten välillä esiintyi yhteneväisyyksiä niin haasteiden kuin niiden mitigoinnin puolesta, mutta myös osa haasteista esiintyi vain satunnaisissa yrityksissä. Yleisesti aineistoista tuli ilmi, että mikropalveluiden ja DevOpsin implementointiin tarvitaan laadukkaita ohjesääntöjä, joiden avulla koko yritys on sisäisesti organisoitunut. Aineistojen yritykset ovat kuitenkin kaikki uniikkeja ja vaikka tietynlaiset ohjesäännöt toimivat heille, ei se takaa sitä toisissa yrityksissä, jolloin joillekin yrityksille ohjesäännöistä ei ole läheskään yhtä suurta hyötyä kuin toisille. Tämän lisäksi aineistojen tutkimukset perustuvat suurimmilta osin kehittäjien ja yritysten muiden työntekijöiden mielipiteisiin ja heidän kokemuksiinsa. Tällöin tuloksiin voi vaikuttaa yksittäisen kehittäjän mielipide ja kokemukset tapahtumasta, eivätkä kyseiset haasteet päde jokaiseen kehittäjään. Vaikka aineistoissa esiintyi yritysten välillä samankaltaisia haasteita ja mitigointistrategioita, ei kuitenkaan nostettu esille, onko kyseiset löydökset yleistettävissä.

Laadukkaita ohjesääntöjä ja suunnittelemista, milloin palvelut ovat tarpeellinen ratkaisu, ehdotettiin palveluiden välisten haasteiden mitigointiin. Aineistoissa ei kuitenkaan nostettu lainkaan palveluiden koon hallitsemista. Monoliittisten sovellusten monimutkaisuus kasvaa aina, kun monoliittiin lisätään uusi osa. Mikropalveluiden yksi hyöty oli pienet palvelut, joita oli helpompi hallita. Kuitenkin jos palveluihin lisätään päättömästi aina uusia osia, tulevat yksittäiset palvelut ajan myötä kasvamaan monimutkaisiksi suuriksi sovelluksiksi. Aineistoissa ei nostettu tarkemmin määrittelyä siitä, kuinka palveluiden kokoa pystytään hallitsemaan tai mikä olisi optimaalinen koko palvelulle. Tämän lisäksi aineistoissa ei lainkaan puhuttu kehittäjien omasta vastuusta noudattaa ohjeita. Voidaan pitää mahdollisena tilannetta, jossa kehittäjä pystyisi refaktoroimaan vastaanotettavaa dataa omassa palvelussaan, jolloin palvelun saisi toimimaan, mutta todellisuudessa ongelma on dataa lähettävän palvelun. Tällöin yrityksen laadukkaasti määritellyt rajapinnat muodostuvat turhiksi ja yksittäiseen palveluun tulee ylimääräistä koodia, joka monimutkaistaa toimintaa.

Aineistoissa puhuttiin paljon putkistojen toiminnasta ja kuinka täytyy implementoida enemmän testattavaa sekä kuinka sovelluksen koonti vaikeutuu. Mainittiin virtuaalikoneiden hidastavan putkistoja ja kontittamisen mitigoivan tämän. Ei kuitenkaan käsitelty lainkaan, kuinka putkistojen aika vaikuttaa kehittäjiin ja milloin putkistojen kesto alkaa vaikuttamaan negatiivisesti kehittäjiin. Putkista saadut hyödyt maksimoituvat, kun kehittäjät tekevät pieniä päivityksiä, jotka aktivoivat putkiston usein. Jos putkiston läpäisy kestää huomattavia aikoja, saattavat kehittäjät päivittää sovellusta harvemmin, koska putkiston läpäisemiseen kuluva aika vie itse kehitykseltä huomattavan aikamäärän, jolloin putkistojen hyödyt menetetään.

Kaikki edellä mainitut haasteet ovat mitigoitavissa sopivissa olosuhteissa, mutta tulee myös miettiä, onko kaikkien haasteiden mitigointi tarpeellista. Varsinkin julkaisussa esiin nostettu tietokannan päivittäminen on ongelma, jota esiintyy vain

ajoittain ja sen mitigointi on hyvin haastavaa ja paljon resursseja käyttävää. Aineistoissa pyrittiin mitigoimaan eteen tulleet haasteet tai todettiin ne ratkaisemattomiksi. Kuitenkaan mitään niistä ei todettu tarpeettomiksi ratkaista tai mietitty tarkemmin mitigoimiseen kuluvia resursseja. Mielestäni olisi tarpeellinen aihe mikropalveluiden ja DevOpsin implementoinnissa tarkistella, mitkä ongelmat kuluttavat enemmän resursseja ratkaisemiseen kuin mitä siitä saataisiin hyötyjä ja mitkä DevOpsin vaiheet ovat kriittisimmät.

4 Yhteenveto

DevOpsia ja mikropalveluarkkitehtuuria verrattiin tutkielmassa enimmäkseen perinteiseen ohjelmistokehitykseen ja monoliittisiin sovelluksiin. DevOpsin hyödyistä nostettiin esille, kuinka se mahdollistaa tiheämmän julkaisutahdin. Tämä mahdollistetaan DevOpsin silmukalla, johon kuuluuvat tärkeät jatkuvan integraation ja jatkuvan julkaisemisen putkistot. Mikropalveluarkkitehtuuri taas helpottaa DevOpsin implementointia, verrattuna suuriin monoliitteihin. Tämän lisäksi mikropalvelut mahdollistavat muita hyötyjä, jotka eivät ole monoliiteissa mahdollisia. Hyötyihin lukeutuu mahdollisuus käyttää eriäviä teknologioita sovelluksen sisällä, virheiden hallitseminen kaatamatta koko sovellusta ja helpompi skaalautuminen.

Ensimmäiseen tutkimuskysymykseen TK1 **"Mitä haasteita mikropalveluiden yhdistäminen DevOpsiin tuo?"** vastattiin tutkielmassa antamalla haasteita kattavasti eri ohjelmistokehityksen vaiheista. Haasteita muodostui lisääntyneistä palveluista, joissa haasteita loivat palveluiden väliset rajapinnat, suorituskyky, yleiskustannukset ja palveluiden hallitsematon käyttäminen. Tämän lisäksi tutkielmassa esitettiin haasteita jatkuvan integraation ja jatkuvan julkaisemisen putkistoiden koonnissa, testaamisessa ja julkaisemisessa. Haasteiksi nousi testaamisen määrän kasvaminen, testaamisen monimutkaisuuden lisääntyminen palveluiden välisistä yhteyksistä, uudet häiriökohdat, kasvaneet riippuvuudet, rajapintojen päivittäminen, tietokannan päivittäminen ja julkaisemisen hitaus. Näiden ongelmien lisäksi tutkielmassa nousi vielä esille haasteita ohjelmistokehityksen ulkopuolelta, joihin lukeutuu

yritykselle kasvavat yleiskustannukset, kehittäjältä vaaditut lisätaidot, muiden kuin kehittäjien kouluttaminen, yrityksen rakenteelliset muutokset ja luottamus.

Toiseen tutkimuskysymykseen TK2 **"Kuinka näitä haasteita hallitaan?"** esitettiin tutkielmassa keinoja, joilla kyseisiä haasteita ollaan pystytty mitigoimaan tai kuinka niiltä pystyttäisiin välttymään. Aineistoista nostettiin jatkuvasti esille yrityksen sisäisiä ohjesääntöjä, joita tulisi luoda ja noudattaa. Näihin ohjesääntöihin suositeltiin lisättäväksi esimerkiksi rajauksia rajapintojen käytöstä. Ohjesääntöjen lisäksi suositeltiin uusien palveluiden ja teknologioiden lisäämiseen prosessia, jolla hallitaisiin lisäyksiä sovellukseen. Aineistoissa painotettiin myös DevOpsin suunnittelua ja monitorointia, joiden avulla pystyttäisiin hallitsemaan erilaisia haasteita, kuten testaamista. Tämän lisäksi kontittaminen korostettiin ratkaisuksi putkistojen haasteisiin. Tutkielmassa todettiin myös joitakin haasteita vielä ratkomattomaksi. Varsinkin tietokannan päivittäminen on ongelma, johon ei aineistosta löytynyt toimivaa ratkaisua.

Tutkielman pohdinnassa nostettiin esille ongelmia, joita voi mahdollisesti muodostua, mutta niistä ei ollut aineistossa lainkaan mainintoja. Näihin haasteisiin lukeutuu palveluiden koon hallitseminen, kehittäjien vastuu toteutuksessa, putkistojen kesto, yleinen tarve mitigoida haasteita ja haasteiden yleistettävyys. Kyseiset ongelmat vaatisivat enemmän yrityksiä, joiden implementointia. Pohdinnasta voidaan esittää seuraavat neljä tutkimuskysymystä:

- Mikä on yksittäisen mikropalvelun optimaalinen koko?
- Kuinka putkistojen kesto vaikuttaa kehittäjiin?
- Milloin DevOpsin haasteiden mitigointi ei ole kannattavaa?
- Mitkä DevOpsin haasteista ovat yleistettävissä?

Kyseiset tutkimuskysymykset esittävät aineistoista esiintyviä ongelmia ja vastaamattomia kysymyksiä, mitä voitaisiin lähitulevaisuudessa tutkia.

Lähdeluettelo

[1] M. Virmani, ”Understanding DevOps & bridging the gap from continuous integration to continuous delivery”, en, teoksessa *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, Galcia, Spain: IEEE, toukokuu 2015, s. 78–82, ISBN: 978-1-4673-7551-1. DOI: `10.1109/INTECH.2015.7173368`. (viitattu 18. 02. 2024).

[2] C. Ebert, G. Gallardo, J. Hernantes ja N. Serrano, ”DevOps”, en, *IEEE Software*, vol. 33, nro 3, s. 94–100, toukokuu 2016, ISSN: 0740-7459. DOI: `10.1109/MS.2016.68`. (viitattu 18. 02. 2024).

[3] D. Trihinas, A. Tryfonos, M. D. Dikaiakos ja G. Pallis, ”DevOps as a Service: Pushing the Boundaries of Microservice Adoption”, en, *IEEE Internet Computing*, vol. 22, nro 3, s. 65–71, toukokuu 2018, ISSN: 1089-7801. DOI: `10.1109/MIC.2018.032501519`. (viitattu 18. 02. 2024).

[4] GitLab Inc., *The GitLab 2022 Global DevSecOps Survey*, en, 2022. url: `https://learn.gitlab.com/dev-survey-22/2022-devsecops-report` (viitattu 18. 02. 2024).

[5] S. R. Dileepkumar ja J. Mathew, ”Enhancing DevOps and Continuous Integration in Software Engineering: A Comprehensive Approach”, en, teoksessa *2023 Second International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT)*, Trichirappalli, India: IEEE, huhti-

kuu 2023, s. 01–05. DOI: 10.1109/ICEEICT56924.2023.10157286. (viitattu 26. 02. 2024).

[6] N. Azad, ”Understanding DevOps critical success factors and organizational practices”, en, teoksessa *Proceedings of the 5th International Workshop on Software-intensive Business: Towards Sustainable Software Business*, Pittsburgh Pennsylvania: ACM, toukokuu 2022, s. 83–90, ISBN: 978-1-4503-9302-7. DOI: 10.1145/3524614.3528627. (viitattu 18. 02. 2024).

[7] M. Senapathi, J. Buchan ja H. Osman, ”DevOps Capabilities, Practices, and Challenges: Insights from a Case Study”, en, teoksessa *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, Christchurch New Zealand: ACM, kesäkuu 2018, s. 57–67, ISBN: 978-1-4503-6403-4. DOI: 10.1145/3210459.3210465. (viitattu 18. 02. 2024).

[8] J. Xuan, T. Duan, Q. Guo et al., ”Microservice Publishing Technology Based on DevOps Architecture”, en, teoksessa *2021 IEEE 5th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*, Xi’an, China: IEEE, lokakuu 2021, s. 1310–1314, ISBN: 978-1-66541-599-6. DOI: 10.1109/ITNEC52019.2021.9586904. (viitattu 18. 02. 2024).

[9] K. Morris, *Infrastructure as Code: Dynamic Systems for the Cloud Age*. O’Reilly, 2021, ISBN: 978-1-09-811467-1. url: https://books.google.fi/books?id=Wz2KzQEACAAJ.

[10] G. Blinowski, A. Ojdowska ja A. Przybylek, ”Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation”, en, *IEEE Access*, vol. 10, s. 20 357–20 374, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3152803. (viitattu 24. 02. 2024).

[11] U. Chouhan, V. Tiwari ja H. Kumar, ”Comparing Microservices and Monolithic Applications in a DevOps Context”, en, teoksessa *2023 3rd Asian Con-*

ference on Innovation in Technology (ASIANCON), Ravet IN, India: IEEE, elokuu 2023, s. 1–7. DOI: 10.1109/ASIANCON58793.2023.10270721. (viitattu 25. 02. 2024).

[12] V. M. Nino-Martinez, J. Octavio Ocharan-Hernandez, X. Limon ja J. C. Perez-Arriaga, "Microservices Deployment: A Systematic Mapping Study", en, teoksessa *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, San Diego, CA, USA: IEEE, lokakuu 2021, s. 24–33, ISBN: 978-1-66544-361-6. DOI: 10.1109/CONISOFT52520.2021.00016. (viitattu 25. 02. 2024).

[13] X. Zhou, H. Huang, H. Zhang, X. Huang, D. Shao ja C. Zhong, "A Cross-Company Ethnographic Study on Software Teams for DevOps and Microservices: Organization, Benefits, and Issues", en, teoksessa *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, Pittsburgh, PA, USA: IEEE, toukokuu 2022, s. 1–10, ISBN: 978-1-66549-590-5. DOI: 10.1109/ICSE-SEIP55303.2022.9794010. (viitattu 08. 03. 2024).

[14] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps", en, teoksessa *2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA: IEEE, huhtikuu 2018, s. 39–397, ISBN: 978-1-5386-6398-1. DOI: 10.1109/ICSA.2018.00013. (viitattu 25. 02. 2024).

[15] S. Baškarada, V. Nguyen ja A. Koronios, "Architecting Microservices: Practical Opportunities and Challenges", en, *Journal of Computer Information Systems*, vol. 60, nro 5, s. 428–436, syyskuu 2020, ISSN: 0887-4417. DOI: 10.1080/08874417.2018.1520056. (viitattu 08. 03. 2024).

[16] M. Callanan ja A. Spillane, ”DevOps: Making It Easy to Do the Right Thing”, en, *IEEE Software*, vol. 33, nro 3, s. 53–59, toukokuu 2016, ISSN: 0740-7459. DOI: 10.1109/MS.2016.66. (viitattu 25. 02. 2024).

[17] V. Debroy ja S. Miller, ”Overcoming Challenges With Continuous Integration and Deployment Pipelines: An Experience Report From a Small Company”, en, *IEEE Software*, vol. 37, nro 3, s. 21–29, toukokuu 2020, ISSN: 0740-7459. DOI: 10.1109/MS.2019.2947004. (viitattu 08. 03. 2024).