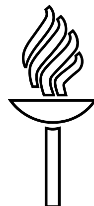
Olli Ketola

MIKROPALVELUARKKITEHTUURIIN PERUSTUVIEN JÄRJESTELMIEN KEHITTÄMISTÄ TUKEVAT TEKIJÄT



JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2019

TIIVISTELMÄ

Ketola, Olli
Mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä tukevat tekijät
Jyväskylä: Jyväskylän yliopisto, 2019, 80 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaaja: Seppänen, Ville

Mikropalveluarkkitehtuuri on ohjelmistokehittämisessä hyödynnettävä pilvipohjainen arkkitehtuurityyli, joka perustuu pieniin itsenäisiin verkkoyhteyden yli kommunikoiviin palveluihin. Tämän työn tavoitteena on selvittää mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä tukevia tekijöitä. Työn empiirisessä tutkimuksessa pyritään vastamaan kahteen tutkimuskysymykseen. Mitkä toimintamallit ja prosessit tukevat mikropalveluarkkitehtuuriin perustuvaa kehittämistä ja mitkä tekniset ratkaisut mahdollistavat mikropalveluarkkitehtuurin tehokkaan toiminnan? Kirjallisuuskatsauksen ensimmäisessä luvussa käydään läpi, mistä mikropalveluarkkitehtuuri on saanut alkunsa ja esitellään aiemman tutkimuskirjallisuuden valossa mikropalveluarkkitehtuurin oleellisiksi tunnistettuja osa-alueita. Toisessa luvussa käydään läpi toimintamalleja ja prosesseja, jotka usein liitettään mikropalveluarkkitehtuurin yhteyteen. Kirjallisuuskatsauksen jälkeen esitellään empiirisen tutkimuksen käytännön toteutus ja käydään läpi tutkimuksen tulokset sekä pohditaan, miten tulokset linkittyvät aiempaan tutkimuskirjallisuuteen. Työn empiirinen tutkimus toteutettiin hyödyntäen laadullista teemahaastatteluihin perustuvaa tutkimusmenetelmää. Haastatellut henkilöt olivat ohjelmistokehittäjiä ja ohjelmistoarkkitehteja, joilla oli aiempaa kokemusta mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämisestä ja suunnittelemisesta. Haastattelujen perusteella saadut tulokset olivat hyvin linjassa aiemman tutkimuskirjallisuuden kanssa. Mikropalveluarkkitehtuuriin perustuvien järjestelmien rakentamiseen liittyen tärkeänä pidettiin DDD-periaatteita seuraavaa suunnittelua, mikropalveluiden mahdollistamaa teknologia riippumatonta kehittämistä, REST-periaatteita seuraavaa kommunikointia ja tietovarastojen eriyttämistä. Lisäksi esiin nousi valmiiden pilvialustojen hyödyntäminen, joka mahdollistaa konttien tehokkaan käytön. Mikropalveluiden kehittämisen kannalta oleellisiksi toimintamalleiksi tunnistettiin DevOps-periaatteita seuraavien käytänteiden omaksuminen ja automaattisten julkaisuprosessien rakentaminen.

Asiasanat: Mikropalvelu, Mikropalveluarkkitehtuuri, DevOps

ABSTRACT

Ketola, Olli
Supporting Factors in Microservice Architecture Development
Jyväskylä: University of Jyväskylä, 2019, 80 pp.
Information Systems, Master's Thesis
Supervisor: Seppänen, Ville

Microservice architecture is an architectural style that is being used for building cloud native software. It is based on small services that communicate through an Internet connection and can be developed and released independently. The goal of this thesis is to identify how microservice based applications are developed at the moment. More precisely, the goal is to investigate what process and methods are used to support microservice development and what kind of technological solutions enables microservice architecture to work efficiently. The first part of the literature review explains where microservice architecture has come from and what is considered the main elements in microservice architecture. In the second part of the literature goes through the processes and the methods that are being used to develop microservice based applications. Rest of the chapters are related to the empirical part of the thesis and discuss how research was done and presents the results of the thesis and discuss how results are in line with previous literature. The empirical part of the thesis was done as a qualitative research by following the theme interview method. Interviewees were software developers and software architects who had experience on building and designing microservice based applications. Results were in line with existing literature. The most significant findings related to the developing microservice based application were, a design that follows DDD principles, technologically independent development, inner communication that follows REST principles, differentiated data storages and extensive use cloud platforms that enables the use of containers. From process and methods perspective DevOps principles and continuous integration and delivery practices were identified to be an integral part of the microservice architecture.

Keywords: Microservice, Microservice Architecture, DevOps

KUVIOT

KUVIO 1 SOA-palvelun jakaminen mikropalveluiksi.. 13
KUVIO 2 Rajapinta asiakkaiden ja mikropalveluiden välillä. 15
KUVIO 3 Vaihtoehdot palvelurekisterin toteuttamiseen 19
KUVIO 4 Mikropalveluiden julkaiseminen konttien avulla. 26
KUVIO 5 Julkaisemisen eri vaiheet 30

TAULUKOT

TAULUKKO 1 DevOpsiin liittyvät periaatteet 34
TAULUKKO 2 Miten DevOps mikropalveluarkkitehtuuri tukevat toisiaan. 41
TAULUKKO 3 Tutkimuksessa haastatellut henkilöt.. 46
TAULUKKO 4 Analyysin tuloksena tunnistetut luokat...................................... 48

SISÄLLYS

TIIVISTELMÄ
ABSTRACT
KUVIOT
TAULUKOT

TIIVISTELMÄ 2

ABSTRACT 3

KUVIOT 4

TAULUKOT 4

SISÄLLYS 5

1 JOHDANTO 7

2 MIKROPALVELUARKKITEHTUURI 9
2.1 Mikropalveluarkkitehtuurin määrittely 10
2.2 Palvelukeskeinen arkkitehtuuri ja mikropalveluarkkitehtuuri 11
2.3 Mikropalveluarkkitehtuurin rakenne 13
2.3.1 Mikropalveluiden välinen kommunikointi 14
2.3.2 Datan hallinta 17
2.3.3 Löydettävyys 18
2.3.4 Skaalautuminen 20
2.3.5 Monitorointi ja virheiden käsittely 21
2.4 Mikropalveluarkkitehtuurin toteuttaminen 22
2.4.1 Mikropalveluiden suunnittelminen 22
2.4.2 Mikropalveluiden kehittäminen 24
2.4.3 Järjestelmien muuntaminen mikropalvelupohjaiseksi 24
2.4.4 Julkaiseminen 25
2.5 Yhteenveto mikropalveluarkkitehtuurista 27

3 MIKROPALVELUIDEN KEHITTÄMISTÄ TUKEVAT TOIMINTAMALLIT JA PROSESSIT 29
3.1 Julkaisemisen hallintaprosessit 30
3.1.1 Jatkuva integraatio 31
3.1.2 Jatkuva toimittaminen 32
3.1.3 Jatkuva julkaiseminen 32
3.2 DevOps 33
3.2.1 Organisaatio ja kulttuuri 35
3.2.2 Automaatio 36

3.3 Tutkimuksia DevOpsin hyödyntämisestä 37
3.4 DevOps mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämisessä 38

4 TUTKIMUKSEN TOTEUTUS 42
4.1 Laadullinen tutkimusmenetelmä 42
4.2 Teemahaastattelu 44
4.3 Aineiston kerääminen 46
4.4 Analysointimenetelmä 47

5 TUTKIMUKSEN TULOKSET 50
5.1 Mikropalveluiden määritelmä 50
5.2 Julkaisuprosessit 50
5.3 Kehittämistä tukevat toimintamallit 51
5.4 Mikropalveluiden muodostaminen 52
5.5 Mikropalveluiden kehittäminen 53
5.6 Mikropalveluiden välinen kommunikointi 54
5.7 Datan hallinta 55
5.8 Virheiden käsittely ja monitorointi 56
5.9 Skaalautuvuus ja löydettävyys 57

6 POHDINTA 59
6.1 Julkaisuprosessit 60
6.2 Kehittämistä tukevat toimintamallit 61
6.3 Mikropalveluiden muodostaminen 62
6.4 Mikropalveluiden kehittäminen 62
6.5 Mikropalveluiden välinen kommunikointi 63
6.6 Datan hallinta 64
6.7 Monitorointi ja virheiden käsitteleminen 65
6.8 Skaalautuminen ja löydettävyys 66
6.9 Jatkotutkimusehdotukset 66

7 YHTEENVETO 67
7.1 Tutkimukseen liittyvät rajoitteet ja menetelmien arviointi 68
7.2 Tulevaisuuden näkymät 69

LÄHTEET 70

LIITE 1 HAASTATTELURUNKO 78

1 JOHDANTO

Organisaatiot tavoittelevat yhä nopeampia kehityssyklejä sekä joustavia ja helposti skaalautuvia järjestelmiä. Tätä tarvetta täyttämään on viime vuosien aikana noussut mikropalveluarkkitehtuuriksi kutsuttu arkkitehtuurityyli. Mikropalveluarkkitehtuurissa järjestelmä jaetaan pieniin itsenäisiin verkkoyhteyden yli kommunikoiviin palveluihin (Newman, 2015). Näin pystytään korvaamaan monoliittiseen arkkitehtuuriin perustuvat järjestelmät, jotka koostuvat ohjelmakooditasolla toisiinsa kytketyistä komponenteista. Mikropalveluihin perustuva kehittäminen on saanut alkunsa suurten teknologiayhtiöiden alettua muuntaa järjestelmiään paremmin skaalautuviksi tukeakseen kasvavia käyttäjämääriä (Jamshidi, Pahl, Mendonca, Lewis, & Tilkov, 2018).

Viime vuosina mikropalveluarkkitehtuuri on kuitenkin yleistynyt myös siihen liitettyjen muiden ominaisuuksien vuoksi, jonka ansioista sitä sovelletaan nykyään myös pienemmissä organisaatioissa. Aiheen tutkiminen on mielekästä ja tärkeää, sillä kasvanut kiinnostus mikropalveluarkkitehtuuria kohtaan on johtanut siihen, että yhä useampi organisaatio toteuttaa järjestelmänsä mikropalveluarkkitehtuuria seuraten (Balalaie, Heydarnoori & Jamshidi, 2016). Mikropalveluarkkitehtuurin suosioita kuvaa hyvin esimerkiksi määritelmä, jonka mukaan siitä on muodostunut standardi tapa toteuttaa pilvipohjaisia järjestelmiä (Balalaie, Heydarnoori, Jamshidi, Tamburri & Lynn, 2018). Mikropalveluarkkitehtuuri näyttelee myös merkittävää roolia vanhojen järjestelmien modernisoinnissa, jota pidetään mikropalveluarkkitehtuuriin yhtenä keskeisenä sovelluskohteena (Bucchiarone, Mazzara, Dustdar, Dragon Larsen, 2018; Jamshidi, ym., 2018). Vaikkakin mikropalveluarkkitehtuuri on nopeasti yleistynyt, siihen liittyvät määritelmät antavat paljon vapauksia sen suhteen, mitä voidaan kutsua mikropalveluarkkitehtuuriksi. Tämän vuoksi mikropalveluarkkitehtuuria myös toteutetaan hyvin eri tavoin. Näin ollen on tärkeää pyrkiä selvittämään, miten mikropalveluarkkitehtuuriin perustuvia järjestelmiä tällä hetkellä kehitetään.

Mikropalveluarkkitehtuuriin perustuvat järjestelmät ovat usein tiukasti sidoksissa sitä tukevaan infrastruktuuriin, joka mahdollistaa itsenäisten mikropalveluiden julkaisemisen ja skaalautumisen. (Claps, Svensson, & Aurum, 2015;

Balalaie ym, 2016; Dragoni ym., 2018). Näin ollen on myös tärkeää selvittää, millaisten toimenpiteiden avulla voidaan tehostaa mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä ja julkaisemista. Tässä tutkielmassa pyritään vastaamaan seuraaviin tutkimuskysymyksiin:

- Millä toimintamalleilla ja prosesseilla voidaan tehostaa mikropalveluarkkitehtuuriin perustuvaa kehittämistä?

- Minkälaisiin ratkaisuihin mikropalveluarkkitehtuuria hyödyntävät järjestelmät perustuvat?

Kirjallisuuskatsauksen avulla pyritään tunnistamaan yllä esiteltyjen tutkimuskysymysten kannalta oleelliset teemat. Työn kirjallisuuskatsaus jakautuu kahteen pääteemaan, joissa käsitellään mikropalveluarkkitehtuuria sekä mikropalveluarkkitehtuurin käyttöä tukevia toimintamalleja ja prosesseja. Tässä työssä käytettävä lähdemateriaali koostuu pääosin konferenssijulkaisuista ja tieteellisten aikakausilehtien artikkeleista. Materiaalina käytetään myös muutamia merkittävinä pidettäviä kirjoja sekä verkkojulkaisuja, joihin myös useat tieteelliset artikkelit viittaavat. Tieteellisten lähteiden keräämisessä hyödynnetiin seuraavia tunnettuja tietokantoja ja hakupalveluja:

- IEEE Xplore Digital Library
- Google Scholar
- Springer Link
- Researchgate
- Wiley Online Library
- ACM Digital Library

Materiaalin keräämisessä käytettiin aiheen kannalta merkittäviä avainsanoja. Mikropalveluarkkitehtuuria käsittelevää materiaalia etsittiin hakusanoilla: *microserivces, microservice, microservice architecture.* Toimintamalleihin ja julkaisuprosesseihin liittyvää materiaalia etsittiin hakusanoilla*: DevOps, continuous integration, continuous delivery, continuous deployment ja release engineering*. Mikropalveluarkkitehtuuria käsittelevä lähdemateriaali koostuu vuosina 2014–2019 tapahtuneista julkaisuista. Tämä osaltaan myös kertoo siitä, kuinka tuoreesta ilmiöstä on kyse.

Kirjallisuuskatsauksen jälkeen esitellään, miten työn empiirinen tutkimus toteutettiin. Työssä hyödynnettiin teemahaastatteluja, jotka toteutettiin haastattelemalla mikropalveluarkkitehtuuriin perustuvia järjestelmiä suunnitelleita ja kehittäneitä henkilöitä. Haastatteluissa hyödynnettiin kirjallisuuskatsauksen avulla mikropalveluarkkitehtuurin kannalta oleellisiksi tunnistettuja teemoja, jotka toimivat haastattelujen pohjana. Myös kerätyn datan analysoinnissa hyödynnettiin kirjallisuuskatsauksen avulla tunnistettuja teemoja. Tutkimuksen ja datan analysoinnin jälkeen, esitellään tutkimuksen tulokset ja lopuksi pohditaan miten tulokset linkittyvät aiempaan tutkimusmateriaaliin.

2 MIKROPALVELUARKKITEHTUURI

Perryn ja Wolfin (1992) mukaan arkkitehtuuri on yksi ohjelmistokehittämiseen liittyvistä osa-alueista. Ohjelmistokehittämisessä arkkitehtuuri kuvaa miten prosessointiin, dataan ja yhteyksiin liittyvät elementit toteuttavat kokonaisuuden, jota voidaan kutsua ohjelmistoksi. Prosessointi määrittelee ohjelmiston toiminnollisuuden ja data kuvaa informaatiota, jota prosessoinnissa käsitellään. Yhteydet määrittelevät miten dataa liikutellaan prosessointia toteuttavien elementtien välillä. Yksinkertaisesti sanottuna ohjelmistojen kehittämiseen hyödynnettävät arkkitehtuurityylit ovat formaaleja kuvauksia siitä, miten data, prosessointi ja yhteydet ovat vuorovaikutuksessa keskenään (Perry & Wolf, 1992).

Mikropalveluarkkitehtuuri on arkkitehtuurityyli, joka pyrkii ratkaisemaan useita ongelmia eriyttämällä järjestelmän sisäiset komponentit toisistaan irrallaan toimiviksi palveluiksi. On hyvin vaikea yksiselitteisesti määritellä kuka tai ketkä ovat arkkitehtuurityylin kehittäjiä. Termi on saanut alkunsa ohjelmistoalan konferensseissa ja arkkitehtuurityylin kehitys on ollut myös pääsääntöisesti ohjelmistoteollisuuden johdattamaa tutkimuksen yrittäessä pysyä perässä (Fowler & James, 2014). Voidaan kuitenkin todeta, että mikropalveluarkkitehtuurin perustuvien järjestelmien yleistyminen on saanut alkunsa yritysmaailman tarpeesta korvata massiiviset monoliittiset ja palvelukeskeiseen arkkitehtuuriin perustuvat järjestelmät, joiden päivittäminen, julkaiseminen ja ylläpitäminen on muuttunut ajan kuluessa hankalaksi (Taibi, Lenarduzzi & Janes, 2017; Namiot & Sneps-Sneppe, 2014; Dragoni ym., 2017).

Mikropalveluarkkitehtuurin hyötyjä usein korostetaan vertaamalla sitä monoliittiseen arkkitehtuuriin. Tyypillisesti monoliittiset järjestelmät perustuvat yhteen suureen kokonaisuuteen, jonka sisäisiä komponentteja ei voida irrottaa toisistaan, kun taas mikropalveluarkkitehtuurin ydinidea on jakaa kehitettävä järjestelmä pieniin itsenäisiin palveluihin, jotka kommunikoivat verkkoyhteyttä hyödyntäen (Newman, 2015, 3).

Mikropalveluarkkitehtuurityylin ensimmäisiä soveltajia ovat olleet suuret teknologiayhtiöt, kuten Netflix, Amazon ja Spotify. Esimerkiksi Spotifyn mikropalveluarkkitehtuuriin perustuvaan musiikin suoratoistopalveluun kuului jo

vuonna 2015 yhteensä 810 erillistä mikropalvelua, joiden kehittämisestä ja ylläpitämisestä vastasi yli 600 kehittäjää (Goldsmith, 2015). Monet suuret yritykset ovat myös olleet hyvin avoimia mikropalveluarkkitehtuuriensa rakenteesta ja julkaisseet useita avoimeen lähdekoodiin pohjautuvia työkaluja, joiden avulla mikropalveluiden kehittämistä, julkaisua ja hallinnointia voidaan edesauttaa (Jamshidi ym., 2018).

Pahlin ja Jamshidi (2016) suorittivat katsauksen mikropalveluarkkitehtuuria käsittelevän tutkimuksen nykytilasta. Tuloksissaan he totesivat tutkimuksen olevan vielä alkutekijöissään ja suurimman osan tieteellisistä julkaisuista keskittyvän mikropalveluiden julkaisuprosessin hallintaan sekä arkkitehtuurityylin määrittelemiseen ja analysoimiseen. Nousevina trendeinä mainittiin mikropalveluiden testaus, hallinnointi, automaatio, monitorointi ja DevOps. Lisäksi mikropalveluarkkitehtuuria kuvailtiin ennemmin kokonaisvaltaiseksi tavaksi rakentaa ohjelmistoja, kuin pelkäksi arkkitehtuurityyliksi, joka kuvaa järjestelmän toimintaa. Myös Soldani, Tamburri ja Heuvel (2018) tunnistivat automaatioon perustuvat julkaisukäytänteet tärkeäksi osa-alueeksi mikropalveluarkkitehtuuria. Pahl ja Jamshidi (2016) ehdottavatkin, että mikropalveluarkkitehtuuria tulisi tarkastella kokonaisuutena, johon sisältyvät arkkitehtuurin lisäksi myös kehittämiseen hyödynnettävät prosessit ja toimintamallit. Seuraavissa luvuissa käsitellään tarkemmin, mistä elementeistä mikropalveluarkkitehtuuri tyypillisesti rakentuu ja mitkä ovat mikropalveluarkkitehtuurille tyypillisiä ominaisuuksia.

2.1 Mikropalveluarkkitehtuurin määrittely

Mikropalveluarkkitehtuuriin liittyvät määritelmät ovat hyvin abstrakteja, eivätkä ne kuvaile tarkkaan, miten mikropalveluarkkitehtuuriin perustuva järjestelmä tulisi toteuttaa. Mikropalveluarkkitehtuuri perustuu osittain vanhoihin ideoihin, joita on aiemmin sovellettu palvelukeskeiseen arkkitehtuuriin perustuvissa järjestelmissä. Fowlerin ja Lewisin (2014) mukaan mikropalvelut kommunikoivat yksinkertaisten verkon yli toimivien rajapintojen kautta, jokainen palvelu vastaa tarkkaan rajatusta määrästä liiketoimintalogiikkaa ja palvelut ovat suunniteltu palautumaan virhetilanteista. Newman (2015) puolestaan määrittelee mikropalvelut pienikokoisiksi autonomisiksi, skaalautuviksi ja vikasietoisiksi palveluiksi, joita voidaan muokata ja päivittää itsenäisesti.

Zimmermann (2017) esittelee katsauksessaan kaikkein kattavimman määritelmän, joka pohjautuu mikropalveluarkkitehtuuria käsitteleviin aiempiin tutkimuksiin. Hänen mukaansa, mikropalveluita voidaan kehittää, julkaista ja skaalata itsenäisesti. Palvelut kommunikoivat tyypillisesti HTTP-protokollaan perustuvien rajapintojen kautta tai hyödyntäen viestijonoja. Mikropalveluarkkitehtuurin suunnittelussa käytetään DDD-suunnitteluperiaatteita (Domain Driven Design) (Evans, 2003), joiden avulla mikropalveluiden toiminnollisuus voidaan rajata liiketoimintavaatimuksien mukaan. Tämän lisäksi Mikropalvelut sisältävät pilvipalveluille tyypillisiä ominaisuuksia, kuten palveluiden hajautus,

eristetty tila, riippumattomuus muista järjestelmistä ja palveluiden automatisoitu hallinta. Mikropalveluarkkitehtuurissa saman järjestelmän palvelut voidaan myös toteuttaa useilla eri teknologioilla ja niiden julkaisuun ja kehittämiseen hyödynnetään tyypillisesti konttiteknologioita (Zimmermann, 2017). Zimmermann (2017) myös korostaa Mikropalveluarkkitehtuuriin perustuvan kehittämisen pohjautuvan jatkuvaan toimittamiseen, jonka mahdollistajana toimii DevOps.

Jamshidin ym. (2018) näkemyksen mukaan mikropalveluarkkitehtuurin mahdollistavia tekijöitä ovat palvelukeskeisestä arkkitehtuurista lähtöisin oleviin ajatuksiin perustuva arkkitehtuuri, joka on toteutettu DDD-suunnitteluperiaatteita seuraten. Lisäksi palvelut tulee suunnitella epäonnistumaan, mikä käytännössä tarkoittaa, että ne pystyvät käsittelemään virhetilanteet automaattisesti sekä skaalautumaan niihin kohdistuvan kuorman kasvaessa (Jamshidi ym., 2018).

Yllä esittelyt määritelmät selittävät, miksi juuri suuria käyttäjämääriä ja jatkuvaa saavutettavuutta tavoittelevat teknologiayhtiöt ovat olleet mikropalveluarkkitehtuurityylin soveltajia. Määritelmät antavat myös kuvan siitä, mitkä tekijät erityisesti tukevat mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä. Seuraavissa alaluvuissa käsitellään, miten mikropalveluarkkitehtuuri juontaa juurensa palvelukeskeisestä arkkitehtuurista ja pyritään selvittämään, mitkä ovat mikropalveluarkkitehtuuriin perustuvien järjestelmien toiminnan kannalta oleellisia elementtejä.

2.2 Palvelukeskeinen arkkitehtuuri ja mikropalveluarkkitehtuuri

Palvelukeskeisestä arkkitehtuurista käytetään yleisesti lyhennettä SOA (Service Oriented Architecture). SOA on palveluihin perustuva arkkitehtuurityyli, jossa palveluiden välinen kommunikointi tapahtuu tyypillisesti WSDL (Web Service Definition Language) määrittelyyn perustuvien rajapintojen kautta, hyödyntäen SOAP (Simple Object Access Protocol) -sanomia tiedonvälitysstandardina (Endrei ym., 2004, 31). Moderneissa SOA-toteutuksissa palveluiden väliset rajapinnat käyttävät usein myös tiedon välittämiseen kevyempiä formaatteja, kuten JSON (Javascript Object Notation) ja rajapinnat määritellään esimerkiksi REST -rajapinnoiksi (Representational State Transfer). Mikropalveluarkkitehtuurin katsotaan usein olevan luonnollinen edistysaskel SOA-arkkitehtuurille, jonka vuoksi sitä on kuvailtu oikeaksi tavaksi toteuttaa SOA-arkkitehtuuri (Newman, 2015).

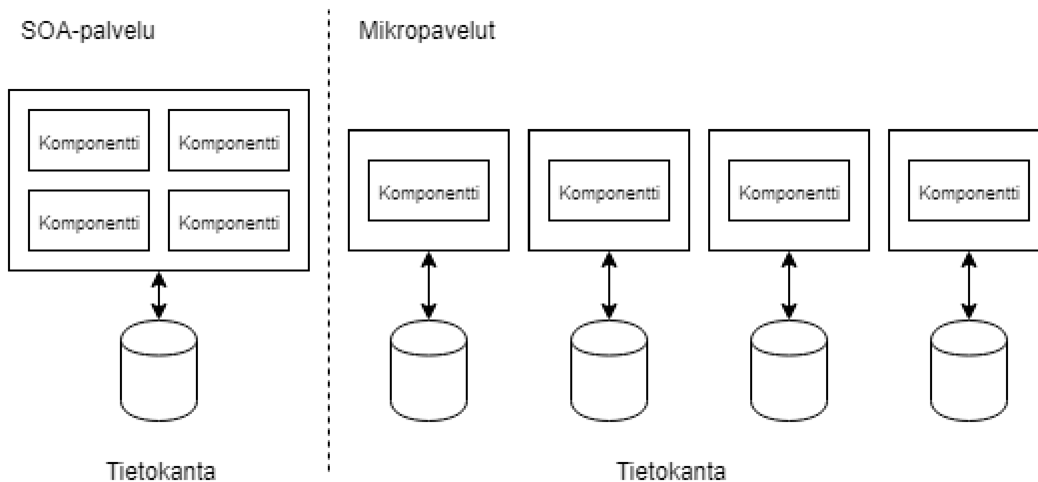
Vaikkakin SOA ja mikropalveluarkkitehtuuri jakavat huomattavan määrän samoja piirteitä, mikropalveluarkkitehtuuria kuitenkin käsitellään usein tieteellisissä julkaisuissa omana arkkitehtuurityylinään (Namiot & Sneps-Sneppe, 2014; Alshuqayran, Ali & Evans, 2016; Francesco, Lago & Malavolta, 2018; Soldani ym., 2018). Zimmermann (2017) puhuu kuitenkin pelkästään mikropalveluista ja pitää mikropalveluarkkitehtuuria ennemminkin osana SOA-arkkitehtuuria. Zimmermann (2017) perustelee kategorisointia sillä, että mikro-

palvelut itsessään eivät tarjoa mitään uutta verrattuna SOA-arkkitehtuuriin, vaan ovat yksinkertaisesti tarkempaan palveluiden jakamiseen perustuva tapa toteuttaa arkkitehtuurityyliä. Tästä huolimatta termi mikropalveluarkkitehtuuri on vakiinnuttanut asemansa tieteellisessä kirjallisuudessa.

Mikropalveluarkkitehtuuriin ja SOA:an välillä on kuitenkin myös paljon eroja. Richards (2015) korostaa SOA:an ja mikropalveluarkkitehtuurin eroja. Hänen mukaansa SOA-arkkitehtuurissa tavoitteena on jakaa mahdollisimman paljon toiminnallisuutta, kun taas mikropalveluarkkitehtuurissa tavoitteena on eristää keskeiset toiminnot omiksi palveluikseen. Lisäksi erityisesti vanhemmissa SOA toteutuksissa palveluiden väliset rajapinnat ovat usein toteutettu raskaiden palvelusopimusten varaan, jonka vuoksi niiden välinen kommunikointi on tiukasti määriteltyä. Mikropalveluarkkitehtuurissa yksi palvelu puolestaan vastaa tarkkaan rajatusta toiminnollisuudesta ja sen rajapinta on tyypillisesti määritelty käsittelemään ainoastaan yksinkertaisia viestejä (Richards, 2015).

Clark (2016) kuvaileekin mikropalveluarkkitehtuuria sovelluskohtaiseksi suunnittelumalliksi, kun taas SOA voi kattaa organisaation koko yritysarkkitehtuurin toteutuksen. Esimerkiksi suurissa toiminnanohjausjärjestelmissä saattaa olla useita käyttöliittymiä ja sovelluksia, joiden kautta voidaan hallinnoida yrityksen eri toimintoja. Lisäksi SOA-toteutuksissa tavoitteena on uudelleen käyttää mahdollisimman paljon samoja palveluita. Uudelleenkäyttöä painotetaan myös ohjelmistokoodissa komponenttitasolla, jonka vuoksi järjestelmään kuuluvat palvelut voivat jakaa samoja komponentteja. Tällöin palveluiden välille syntyy kiinteitä riippuvuuksia. Tämän lisäksi SOA-järjestelmään kuuluvat palvelut myös tyypillisesti jakavat tietokannan (Clark, 2016). Tämä johtaa siihen, että periaatteessa erillisiin palveluihin perustuva arkkitehtuuri muistuttaakin monoliittista ohjelmaa, eikä sitä voida julkaista tai muuttaa riippumatta muista järjestelmään kuuluvista palveluista (Cerny, Donahoo & Trnka, 2017).

Mikropalveluarkkitehtuurissa tavoitteena on jakaa mahdollisimman vähän (kuvio 1) palveluiden kesken, jolloin palveluiden itsenäisyys pystytään säilyttämään. Mikropalveluarkkitehtuurissa myös jokainen palvelu vastaa siihen liittyvän datan tallentamisesta, jolloin jokaisella mikropalvelulla on tyypillisesti oma tietokantansa (Richardson, 2018, 14).



KUVIO 1 SOA-palvelun jakaminen mikropalveluiksi.

Dragoni ym. (2018) mainitsevat SOA:n ja mikropalveluarkkitehtuurin yhdeksi keskeisimmäksi eroksi skaalautuvuuden. Tyypillisesti SOA-pohjainen palvelu vastaa useista eri toiminnoista, joihin kohdistuva kuorma saattaa vaihdella. Tämän vuoksi järjestelmää skaalattaessa, koko palvelusta tarvitsee luoda aina uusi ilmentymä. Mikropalveluarkkitehtuuri mahdollistaa puolestaan yksinkertaisen ja huomattavasti kevyemmän tavan skaalata järjestelmää, koska skaalautuminen voidaan kohdistaa tarkkaan ainoastaan niihin mikropalveluihin, joihin kohdistuu eniten kuormaa. Cerny ym. (2017) korostavat vertailussaan, miten mikropalveluarkkitehtuuri ja SOA eroavat toisistaan järjestelmän sisällä tapahtuvan kommunikoinnin osalta. SOA-palveluiden välisestä kommunikoinnista vastaa usein ESB (Enterprise Service Bus), joka sisältää palveluiden väliseen integraatioon vaadittavaa logiikkaa, kun taas mikropalveluarkkitehtuurissa mahdollisimman suuri määrä kommunikointiin liittyvästä logiikasta on sijoitettu suoraan palveluun. Esimerkiksi SOA-palvelulla ei välttämättä ole tietoa kenelle sen lähettämä viesti on osoitettu, vaan kommunikointiväylän tehtävänä on reitittää viesti oikealle vastaanottajalle (Cerny ym., 2018).

Kuten voidaan havaita SOA ja mikropalveluarkkitehtuuri jakavat suuren määrän samoja piirteitä, mutta mikropalveluarkkitehtuurin perustuvien järjestelmien toiminta eroaa kuitenkin monilta osin perinteisistä SOA toteutuksista.

2.3 Mikropalveluarkkitehtuurin rakenne

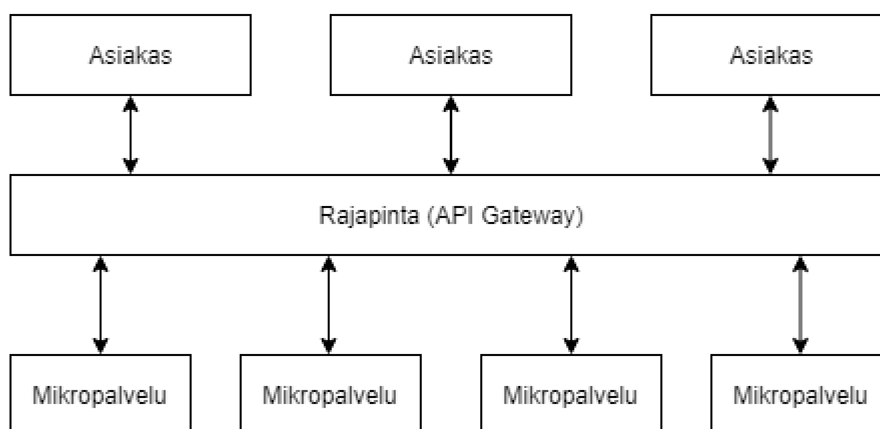
Mikropalveluarkkitehtuuriin perustuvat järjestelmät ovat hajautettuja järjestelmiä, mikä itsessään ei ole uusi tapa kehittää ohjelmistoja. Vaikkakin mikropalveluarkkitehtuurissa hyödynnetään suurelta osin samoja elementtejä, kuin SOA -pohjaisissa järjestelmissä niiden soveltaminen ei kaikissa tilanteissa toimi enää samaan tapaan (Sill, 2016). Suuri määrä pieniä palveluita aiheuttaa uusia haasteita. Näistä keskeisimpiä ovat oikeanlaisen palvelujaon löytäminen, järjestelmään kuuluvien mikropalveluiden välisen kommunikoinnin määritteleminen,

infrastruktuuria ylläpitävät ohjelmat ja palvelut sekä automaatioon perustuvien julkaisuprosessien rakentaminen (Francesco ym., 2018; Soldani ym., 2018).

Mikropalveluarkkitehtuurin kannalta merkittävimpinä ominaisuuksina voidaan pitää palveluiden skaalautuvuutta, ylläpidettävyyttä ja helppoa julkaisemista. Näiden ominaisuuksien tärkeimpänä mahdollistajana toimii mikropalveluiden itsenäisyys, joka mahdollistaa järjestelmään kuuluvien mikropalveluiden riippumattoman kehittämisen ja julkaisemisen. (Newman, 2015, 3; Alshuqayran ym., 2016). Mikropalveluarkkitehtuuriin perustuvien järjestelmien rakenteeseen liittyen on tunnistettu useita suunnittelumalleja, joiden käyttöä tehokas mikropalveluarkkitehtuurin hyödyntäminen edellyttää. Suunnittelumaleja seuraamalla pyritään erityisesti vaikuttamaan mikropalveluiden väliseen kommunikointiin, datan hallintaan, mikropalveluiden löydettävyyteen ja skaalautumiseen sekä monitorointiin ja virheiden käsittelemiseen. (Newman, 2015; Alshuqayran, ym. 2016; Zimmermann, 2017; Taibi, Lenadruzzi & Janes, 2017; Soldani ym., 2018). Seuraavissa alaluvuissa käsitellään mikropalveluarkkitehtuuriin perustuvien järjestelmien rakenteeseen liittyviä tärkeitä osa-alueita.

2.3.1 Mikropalveluiden välinen kommunikointi

Mikropalveluarkkitehtuurissa palveluiden välinen kommunikointi tapahtuu verkon yli toimivien rajapintojen kautta, jotka voivat olla toteutettu useita eri protokollia seuraten. Useissa lähteissä mikropalveluarkkitehtuuriin liittyväksi tärkeäksi suunnittelumalliksi tunnistetaan mikropalveluiden toiminnollisuuksien tarjoaminen yhtenäisen rajapintapalvelun (API Gateway) (kuvio 2) kautta (Namiot & Sneps-Sneppe, 2014; Newman, 2015, 71; Dragoni, ym., 2017; Francesco, Malavolta & Lago, 2017; Taibi ym., 2018). Sisäänkäyntinä järjestelmään toimiva rajapinta usein toteutetaan yksinkertaisena REST-rajapintana ja se välittää kyselyjä järjestelmään kuuluville mikropalveluille ja kokoaa saamansa vastaukset käyttöliittymässä mielekkäällä tavalla esitettävään muotoon. Rajapinnan ansioista järjestelmään kuuluvien mikropalveluiden kanssa kommunikoivien osapuolien ei tarvitse olla tietoisia erillisistä mikropalveluista, vaan kaikki kommunikointi tapahtuu rajapinnan kautta (Taibi ym., 2017). De La Torren Wagnerin ja Rousosin (2019, 48–49) mukaan yhtenäisen rajapintapalvelun avulla pystytään myös toteuttamaan järjestelmään sisään tulevien kyselyiden autentikointiin, kuormantasaukseen ja monitoroitiin liittyviä toimintoja.



KUVIO 2 Rajapinta asiakkaiden ja mikropalveluiden välillä.

Mikropalveluiden eristäminen rajapinnan avulla mahdollistaa myös järjestelmän sisällä tapahtuvan vapaan kommunikoinnin, jolloin esimerkiksi yhteyden salaaminen voidaan suorittaa asiakkaiden ja rajapinta kerroksen välissä (De La Torre ym., 2019, 48–49). Yhtenäisen rajapinnan luomista tukee myös se, että mikropalveluiden ja niitä kutsuvien asiakasohjelmistojen välinen suora kommunikointi on tunnistettu mikropalveluarkkitehtuuriin liittyväksi antisuunnittelumalliksi, joka johtaa asiakkaan ja järjestelmän välisiin suoriin riippuvuuksiin ja aiheuttaa ylimääräistä työtä asiakasohjelmistojen toteuttamisessa (Taibi & Lenardruzzi, 2018). Yhtenäisen rajapinnan haittapuolia ovat sen muodostama pullonkaula järjestelmään saapuvien kutsujen välittäjänä, joka kaatuessaan estää koko järjestelmän toiminnan. Lisäksi rajapinta lisää ylimääräisen verkon ylitse tapahtuvan kyselyn järjestelmän ja asiakkaiden välille (De La Torre ym., 2019, 49).

Mikropalveluarkkitehtuurissa järjestelmän sisäinen kommunikointi voidaan tyypillisesti jakaa synkroniseen- ja asynkroniseen kommunikointiin, jotka perustuvat palveluiden väliseen orkestrointiin ja koreografiaan (Newman, 2015, 42). Orkestrointi ja koreografia termit ovat lähtöisin SOA-arkkitehtuurista, mutta niiden merkitys korostuu erityisesti mikropalveluarkkitehtuurissa järjestelmään kuuluvien palveluiden määrän kasvaessa (Dragoni, ym., 2017). Orkestrointi liitetään usein palveluiden väliseen synkroniseen kommunikointiin, kun taas koreografian kautta kuvataan asynkronisesti tapahtuvaa kommunikointia.

Richardson (2018, 66–68) mukaan mikropalveluiden väliseen orkestrointiin perustuvassa kommunikoinnissa yksi mikropalvelu toimii viestien välittäjänä alemman tason mikropalveluille. Sama mikropalvelu myös kokoaa alemmilta tasoilta saamansa paluuviestit ja välittää ne eteenpäin. Orkestrointiin perustuva kommunikointi on yleisesti tyyliltään synkronista ja perustuu kysely-vastaus (Request Response) -malliin, jossa mikropalvelu saa rajapintansa kautta kyselyn ja vastaa siihen mahdollisimman nopeasti. Tällaiset rajapinnat toimivat usein HTTP-protokollan avulla ja seuraavat REST-periaatteita. REST-periaatteita seuraavan kommunikoinnin sijaan voidaan myös käyttää erilaisia RPC-kyselyjä (Remote Procedure Call), jotka helpottavat monimutkaisten kyse-

lyiden suorittamista. RPC-muotoiset kyselyt perustuvat yleensä tunnettuihin tiedonvälistysformaatteihin kuten JSON ja XML (Richardson, 2018, 66–68). Synkronisen kommunikoinnin keskeisimpiä hyötyjä on sen helppo ymmärrettävyys ja yksinkertainen toteuttaminen (Richardson, 2018, 77–78).

Toinen tapa kommunikoida mikropalveluarkkitehtuurissa on asynkroninen kommunikointi, jossa kyselyn lähettäjä ei tarvitse välitöntä tietoa operaation lopputuloksesta. Richardsonin (2018, 86–87) mukaan asynkronisessa kommunikoinnissa mikropalveluiden välinen tiedonvaihto perustuu niiden välillä olevaan koreografiaan, joka voidaan toteuttaa esimerkiksi julkaisija-tilaaja–suunnittelumallia seuraamalla. Julkaisija-tilaaja -suunnittelumalli koostuu tyypillisesti kolmesta elementistä, jotka ovat julkaisija, viestijono ja tilaaja. Yhden julkaisijan viestejä voivat kuunnella useat tilaajat. Julkaisijan tehtävänä on tuottaa uusia viestejä viestijonoon, jonka kautta tilaajat vastaanottavat viestejä. Viestejä voidaan välittää tilaajille kahden eri menetelmän avulla. Tilaajat joko kysyvät säännöllisin väliajoin itselleen osoitettuja viestejä tai vastaavasti rekisteröityvät vastaanottamaan viestejä automaattisesti niiden saapuessa, jolloin viestijonoa ylläpitävä palvelu työntää viestejä niitä kuuntelemaan rekisteröidyille tilaajille (Richardson, 2018, 86–87). Julkaisija-tilaaja –suunnittelumallin ansiosta useat palvelut voivat käsitellä eri tapahtumiin liittyviä viestejä samaan aikaan.

Viestijonojen toteuttamiseen on useita eri vaihtoehtoja, jonot voidaan rakentaa esimerkiksi siten, että viestit tallennettaan tietokantaan. Tietokantaan perustuvassa toteutuksessa julkaisijana toimivat mikropalvelut tallentavat viestejä tietokantatauluun, johon viestejä tarkkailevat mikropalvelut suorittavat säännöllisen väliajoin kyselyn ja käsittelevät niille osoitetut viestit. Richardson (2018, 93) suosittelee viestijonojen toteuttamiseen valmiita ratkaisuja, kuten RabbitMQ ja Aapache Kafka, joissa viestijonon ylläpidosta vastaa erillinen kolmannen osapuolen toteuttama ohjelmisto, joka asennetaan palvelimelle erikseen. Valmiiden viestijonojen avulla pystytään nopeasti rakentamaan vikasietoisia ja skaalautuvia viestijonoja, jotka kykenevät käsittelemään suuren määrän dataa (Richardson, 2018, 93). Sillin (2016) mukaan asynkronisessa kommunikoinnissa viestien välittämiseen käytetään tyypillisesti viestinvälitysprotokollia, kuten AMQP (Advanced Message Queuing Protocol) ja MQTT (Message Queuing Telemetry Transport).

Asynkronisen kommunikoinnin keskeisiä hyöytyjä ovat viestijonojen kautta tapahtuvan kommunikoinnin avulla saavutettava riippumaton arkkitehtuuri, jonka skaalaaminen on helpompaa, kuin synkroniseen kommunikointiin perustuvassa järjestelmässä. Tämä on mahdollista koska viestijonoon viestejä toimittavien julkaisijoiden ei tarvitse olla tietoisia viestien käsittelijöistä, jolloin viestien käsittelystä vastuussa olevien mikropalveluiden määrää voidaan huoletta kasvattaa tai vähentää riippuen jonossa olevien viestien määrästä (Newman, 2015, 57). Usein kuitenkin mikropalveluarkkitehtuuri koostuu palveluista, joiden halutaan toimivan synkronisesti ja asynkronisesti, jolloin arkkitehtuurissa yhdistellään molempia kommunikointi menetelmiä.

Valituista kommunikointitavoista riippumatta mikropalveluarkkitehtuuri sisältää suuren määrän verkkoyhteyden yli kulkevia operaatioita. Tämän vuoksi mikropalveluiden välisen kommunikoinnin säilyttäminen yksinkertaisena ja minimaalisena muodostuu erityisen tärkeäksi mitä suuremmaksi kehittävä järjestelmä kasvaa.

2.3.2 Datan hallinta

Mikropalveluarkkitehtuurissa datan käsittelemiseen on esitetty suunnittelumallia, jonka mukaan jokaisen datavaraston tarvitsevan mikropalvelun tulisi omistaa oma tietokanta (Taibi ym., 2018; Soldani ym., 2018). Suunnittelumallia tukevat myös Taibin ja Lenadruzzni (2018) löydökset, joiden mukaan jaettujen tietokantojen käyttö tunnistettiin antisuunnittelumalliksi. Jaettuja tietokantoja käyttämällä menetetään mikropalveluiden itsenäisyys, sillä tietokantamalliin tapahtuvat muutokset vaikuttaisivat kaikkiin sitä tietovarastonaan käyttäviin mikropalveluihin (Taibi & Lenadruzzi, 2018).

Tietokantojen erittely tuo kuitenkin mukanaan myös haasteita. Järjestelmän käsittelemät komennot voivat olla yhteydessä useisiin mikropalveluihin. Näin ollen esimerkiksi dataa tallennettaessa syntyy hajautettuja transaktioita, joita pidetään yhtenä mikropalveluarkkitehtuuriin perustuvien järjestelmien keskeisimpinä haasteina (Soldani ym., 2018). Näin ollen mikropalveluarkkitehtuuri aiheuttaa erityisiä haasteita kehitettäessä järjestelmiä, joilta edellytetään ACID-ominaisuuksien (Atomicity, Consistency, Isolation, Durability) mukaista toimintaa, joka takaa, että kaikki loppukäyttäjille palautettava data on aina yhdenmukaista ja sisältää viimeisimmät siihen tapahtuneet muutokset (Nadareishvili, Mitra, McLarty & Amundsen, 2016, 79).

Mikropalveluarkkitehtuuria kuvaillaankin arkkitehtuurityyliksi, joka ei takaa datan yhdenmukaisuutta reaaliaikaisesti, mutta jossa data päätyy aina lopulta yhdenmukaiseen tilaan (Eventually Consistent) (Newman, 2015, 233). Mikropalveluarkkitehtuuriin perustuvat järjestelmät seuraavat tyypillisesti Pritchettin (2008) esittelemiä BASE-ominaisuuksia (Basically available, Soft state, Eventually consistent), joita voidaan pitää vastakohtana ACID-ominaisuuksille. BASE-ominaisuuksien mukaan rakennettu järjestelmä suosii saavutettavuutta yli yhdenmukaisen tila. Tällaisessa järjestelmässä loppukäyttäjille tarjoiltava data ei välttämättä ole yhdenmukaista, mutta päätyy aina lopulta yhdenmukaiseen tilaan (Pritchett, 2008).

Datan hallintaan liittyen mikropalveluarkkitehtuurin yhdeksi suurimmista haasteista on tunnistettu hajautettujen transaktioiden käsitteleminen (Viggiato, Terra, Rocha & Valente, 2018). Rirchardsonin (2018, 112–114) mukaan mikropalveluarkkitehtuurissa hajautettujen transaktioiden hallintaan voidaan soveltaa, joko kaksivaiheista hyväksyntää (Two Phase Commit) tai Saagoja (Sagas). Perinteisesti tietokantajärjestelmä on ollut vastuussa datan yhdenmukaisuuden takaamisesta, mutta mikropalveluarkkitehtuurissa tämä ei kuitenkaan ole aina mahdollista, koska datan tallentaminen voi hajautua useiden tietokantojen kesken. Näin ollen yhdenmukaisen tilan saavuttaminen on varmistettava

komennon käsittelemiseen osallistuvissa mikropalveluissa. Kaksivaiheisen hyväksynnän käyttöä rajoittaa, sen synkronisuus ja näin ollen hidas toiminta sekä sen yhteensopimattomuus modernien NoSQL-tietokantajärjestelmien kanssa (Richardson, 2018, 112–114).

Carcia-Molina ja Salem (1987) ovat esittäneet hajautettujen transaktioiden hallintaan ratkaisuksi Saagoja, joiden avulla kuvataan transaktioon liittyvät askeleet. Saagojen ideana on, että jokaiselle transaktioon kuuluvalle askeleelle on määritetty kompensoiva operaatio, jonka avulla askel voidaan peruuttaa. Mikäli yksikin transaktioon sisältyvä askel epäonnistuu, tulee kaikki tähän mennessä tehdyt askeleet peruuttaa suorittamalla kompensoivat operaatiot jo suoritetuille askeleille. (Carcia-Molina ja Salem, 1987).

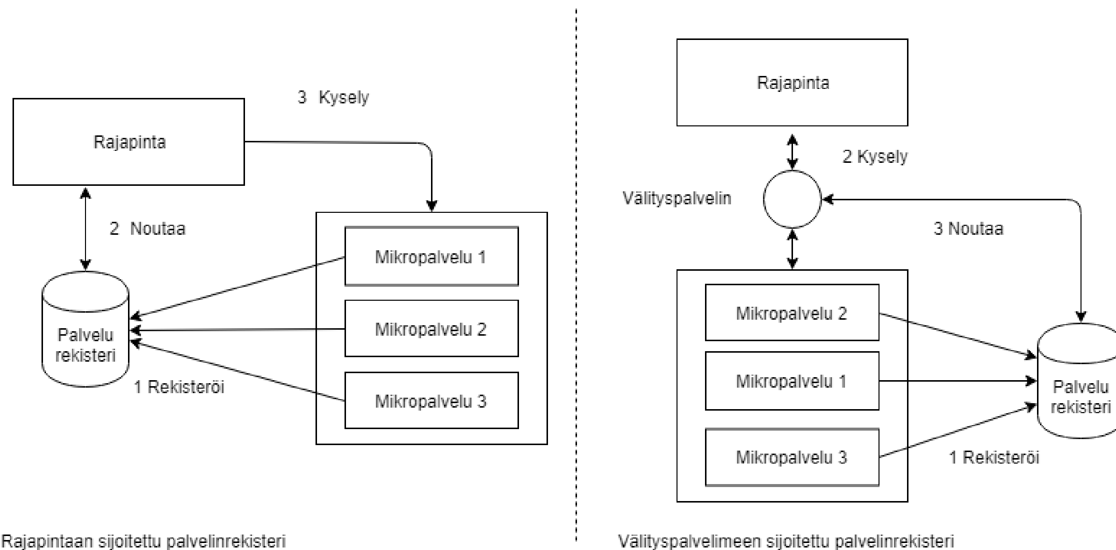
Richardson (2018, 118–125) esittelee saagojen toteuttamiseen kaksi vaihtoehtoa. Ne voidaan toteuttaa hyödyntämällä joko mikropalveluiden välistä orkestrointia tai koreografiaa. Orkestroinnissa transaktion tilaa seuraa erillinen mikropalvelu, joka on yhteydessä kaikkiin transaktiossa mukana oleviin mikropalveluihin. Mikäli yksikin transaktion osallistuva mikropalvelu epäonnistuu tehtävässään, niin orkestroinnista vastaava mikropalvelu lähettää jo tehtävänsä suorittaneille mikropalveluille kompensoivan transaktion. Mikropalveluiden välisessä koreografiassa hajautettujen transaktioiden käsitteleminen perustuu palveluiden välillä tapahtuvien asynkronisten viestien välittämiseen. Mikropalveluiden koreografian avulla toteutettujen hajautettujen transaktioiden hallintaan ei sisälly keskitettyä hallinnointia, jonka avulla ylläpidettäisiin transaktion tilaa, vaan transaktion suorittamiseen osallistuvat mikropalvelut kommunikoivat keskenään viestijonojen kautta. Näin ollen jokaisen transaktioon osallistuvan mikropalvelun on kyettävä vastaanottamaan, ja käsittelemään tietoja epäonnistuneesta transaktioista (Richardson, 2018, 118–125).

2.3.3 Löydettävyys

Mikropalveluarkkitehtuuri mahdollistaa järjestelmän helpon skaalaamisen, mutta se kuitenkin aiheuttaa haasteita palveluiden löydettävyydessä. Esimerkiksi tilanteissa, joissa palvelimelle halutaan automaattisesti lisätä uusi ilmentymä mikropalvelusta, on järjestelmän muiden mikropalveluiden saatava tieto uuden ilmentymän sijainnista. Järjestelmään kuuluvien mikropalveluiden kehitys voidaan myös hajauttaa useiden tiimien kesken, jolloin ilman dynaamista palveluiden löydettävyyttä eri tiimien täytyy koordinoida keskenään, missä osoitteessa mikäkin mikropalvelu toimii. Tuotantoympäristöissä halutaan myös yleensä varmistaa, että järjestelmän eri toimintoja palvelevat mikropalvelut ovat edelleen saavutettavissa, vaikka yhteys osaan palvelimista katkeaisi. Tämän vuoksi mikropalveluarkkitehtuuriin perustuvan järjestelmän yksittäiset palveluinstanssit hajautetaan useille fyysisille palvelimille. Näin ollen löydettävyyden tulee toimia myös eri palvelimille sijoitettujen mikropalveluiden välillä.

Richardson (2018, 82–84) esittelee palveluiden löydettävyyteen liittyen kaksi mahdollista tapaa. Palveluiden sijainnin selvittäminen voidaan tehdä joko kyselyjä järjestelmään välittävän rajapinnan toimesta tai se voidaan suorittaa

järjestelmän sisällä. Molemmissa tapauksissa löydettävyys voidaan rakentaa keskitetyn rekisterin varaan, johon järjestelmään kuuluvat mikropalvelut päivittävät automaattisesti tiedot sijanneistaan ja joista kyselyitä suorittavat tahot noutavat tietoja, ottaakseen yhteyden haluttuun mikropalveluun (Richardson, 2018, 82–84). Rajapintapuolen toteutuksessa (kuvio 3) rekisteriä ylläpidetään järjestelmään kyselyitä suorittavassa rajapinnassa, tässä ratkaisussa myös kyselyiden välittäminen saman mikropalvelun eri instanssien välillä on toteutettu suoraan rajapintaan.



KUVIO 3 Vaihtoehdot palvelurekisterin toteuttamiseen (Richardson, 2018, 82–84).

Palvelinpuolen toteutuksessa (kuvio 3) palveluiden löydettävyydestä vastaa välityspalvelin, joka toimii yhteistyössä palvelurekisteriä ylläpitävän ohjelmiston kanssa. Molemmissa menetelmissä jokaisen järjestelmään kuuluvan mikropalvelun on kyettävä automaattisesti rekisteröimään oma sijaintinsa keskitettyyn palvelurekisteriin. Mikropalveluiden rekisteröinti voidaan toteuttaa esimerkiksi kirjoittamalla jokaiseen mikropalveluun erikseen logiikka automaattista rekisteröitymistä varten. Tyypillisempää on kuitenkin hyödyntää kolmansien osapuolien toteuttamia ohjelmistoagentteihin perustuvia ratkaisuja, joissa jokaisen mikropalvelun yhteyteen julkaistaan agentti, joka osaa välittää siihen liittyvän mikropalvelun tiedot keskitettyyn palvelurekisteriin (Kookarinrat & Temtanapat, 2016; Kang, Le, & Tao, 2016). Erillisiä ratkaisuja palvelurekisterin ylläpitoon ovat esimerkiksi Consul ja Zookeeper (Jamshidi ym., 2018). Erillisten ohjelmistojen sijaan yksi mahdollinen ratkaisu palveluiden löydettävyyden ratkaisemiseen on hyödyntää mikropalveluiden julkaisemiseen käytettävien alustojen sisäänrakennettuja palvelurekisterejä, näitä tarjoavat mm. Docker Swarn ja Kubernetes (Richardson ,2018, 82–84).

2.3.4 Skaalautuminen

Helppoa skaalautuvuutta pidetään mikropalveluarkkitehtuurin yhtenä tärkeimmistä eduista. Useaan itsenäiseen mikropalveluun jaettua järjestelmää pystytään skaalaamaan tarkasti toiminnollisuus kohtaisesti. Näin saavutetaan huomattavia etuja verrattuna SOA -tai monoliittisiin arkkitehtuureihin perustuvien järjestelmien skaalautumiseen. Dragonin ym. (2018) mukaan yksi skaalautumisen tavoitteista on kyetä lisäämän järjestelmän käsittelykapasiteettia, siihen kohdistuvan kuorman kasvaessa. Skaalaamalla järjestelmää voidaan kuitenkin vaikuttaa myös siihen kuuluvien mikropalveluiden saavutettavuuteen. Useiden hajautettujen mikropalveluinstanssien tarkoituksena ei ole välttämättä palvella mahdollisemman montaa asiakasta, vaan niiden avulla voidaan myös turvata järjestelmän toiminta vikatilanteissa (Dragoni, ym., 2018).

Abbottin ja Fisherin (2009, 340) esittelemän mallin mukaan skaalautuminen voidaan toteuttaa kolmella tapaa. 1) Koko järjestelmä voidaan monistaa kokonaisuudessaan usealle eri palvelimelle, jolloin kaikki palvelimet pystyvät toteuttamaan kaikki järjestelmälle mahdolliset operaatiot. 2) Järjestelmään voidaan jakaa useisiin palveluihin, jolloin osiin jaettu järjestelmä hajautetaan useille eri palvelimille. 3) Järjestelmä monistetaan kokonaisuudessaan usealle eri palvelimelle, mutta saapuvat kyselyt voidaan ohjata eri palvelimille, siten että jokainen palvelin käsittelee ainoastaan tietynlaisia kyselyjä (Abbott & Fisher, 2009, 340).

Usein järjestelmien eri osat vaativat erilaista skaalautuvuutta, esimerkiksi joidenkin operaatioiden toteuttaminen saattaa vaatia pitkäkestoista raskasta laskentaa palvelimen prosessilta, kun taas toisten operaatioiden vaatimuksena on käsitellä mahdollisimman suuri määrä tietokantakyselyitä. Mikropalveluarkkitehtuurissa tällainen järjestelmä voidaan hajauttaa toimimaan useilla palvelimilla, joiden ominaisuudet vastaavat mikropalvelulle asetettuja vaatimuksia. Dockerin (2018) kaltaisten työkalujen avulla skaalautuminen pystytään suorittamaan reilaajassa käynnistämällä mikropalvelun sisältämiä kontteja ainoastaan siksi aikaa, kun suurempaa käsittelykapasiteettia tarvitaan. Näin toimimalla pysytään säästämään palvelin kustannuksissa, joiden hinta perustuu palvelimen käyttöasteeseen. Automaattisten skaalautumistoimenpiteiden suosiosta kertoo SysDig (2018) raportti, jonka mukaan konttien keskimääräinen elinikä on vain 5–10 minuuttia.

Newmanin (2015, 220–224) mukaan skaalautuvuuteen voidaan vaikuttaa myös valitsemalla erityisiä suunnittelumalleja järjestelmän kriittisten toimintojen toteuttamiseen. Esimerkkisi suuren määrän tietokantaan tapahtuvia kysely- ja kirjoitusoperaatioita käsitelevän mikropalvelun toteuttamiseen voidaan soveltaa CQRS-suunnittelumallia (Command, Query, Responsibilty, Segregation). CQRS:n avulla skaalautuminen voidaan viedä äärimmilleen jakamalla yksi mikropalvelu kahdeksi erilliseksi mikropalveluksi, joista toinen käsittelee ainoastaan kyselyihin ja toinen tietokantaan kirjoittamiseen liittyvät operaatiot. Tällaista suunnittelumallia seuraamalla kyselyitä käsittelevien mikropalveluiden määrää voidaan kasvattaa riippumatta käskyjä käsittelevästä mikropalveluista.

Toinen helppoa skaalautuvuutta tukeva suunnittelumalli on viestijonojen hyödyntäminen. Viestijonoihin perustuvien järjestelmien skaalaaminen on yksinkertaista, sillä viestejä vastaanottavien mikropalveluiden määrää voidaan kasvattaa riippumatta viestejä välittävästä palvelusta (Newman, 2015, 220–224). Kuten voidaan havaita, mikropalveluihin perustuvia järjestelmiä voidaan skaalata useiden eri menetelmien kautta.

2.3.5 Monitorointi ja virheiden käsittely

Mikropalveluarkkitehtuurin hajautetun luonteen vuoksi virheiden käsittely aiheuttaa ylimääräisiä haasteita. Newmanin (2015, 205) mukaan mikropalveluarkkitehtuuriin perustuvaa järjestelmään suunniteltaessa on seurattava ajatusmallia, jonka mukaan kaikki mikä voi epäonnistua, tulee jossain vaiheessa epäonnistumaan. Näin ollen erityistä huomiota tulisi kiinnittää ennemminkin mahdollisimman tehokkaaseen virheistä palautumiseen, kuin pyrkiä estämään jokainen mahdollinen virhetilanne (Newman, 2015, 215–216). Myös Klepmann (2017, 227) kuvailee hajautettuja järjestelmiä epäluotettaviksi järjestelmiksi, jotka koostuvat useista elementeistä, jotka tulevat jossain vaiheessa järjestelmän elinkaarta epäonnistumaan.

Näin ollen järjestelmään kuuluvien mikropalveluiden monitorointi ja virheiden käsittely ovat osa-alueita, joiden avulla voidaan parantaa mikropalveluarkkitehtuurin perustuvan järjestelmän toimintavarmuutta. Molemmat osa-alueet liittyvät myös oleellisesti järjestelmän skaalautuvuuteen, sillä usein skaalautumisen tarpeen aiheuttaa järjestelmässä tapahtuva odottamaton virhetilanne tai monitoroinnin avulla havaittava suuremman käsittelykapasiteetin tarve. Monitoroinnin tehtävänä on tarkkailla järjestelmään kuuluvien palvelimien tuottamia tietoja esimerkiksi yhteyksien, muistin ja keskusyksiköiden käyttöön liittyen. Lisäksi monitorointi tarkkailee jokaiseen järjestelmään kuuluvan mikropalvelun tuottamia lokitietoja, jotka voivat sisältää tietoa mm. yksittäiseen mikropalveluun kohdistuvasta kuormasta ja virhetilanteiden määrästä. Näiden tietojen yhdistelmällä voidaan luoda kuva järjestelmään kohdistuvasta skaalautumistarpeesta, joka välitetään skaalautumista hoitavalla palvelulle.

Mikropalveluiden väliseen kommunikointiin liittyvät virhetilanteet ovat tunnistettu useissa lähteissä yhdeksi keskeiseksi haasteeksi mikropalveluarkkitehtuurissa (Balalaie, Heydarnoori & Jamshidi, 2016; Jamshidi ym., 2018; Soldani, 2018; Balalaie, ym., 2018). Esimerkki tyypillisestä hajautettuihin järjestelmiin liittyvästä virhetilanteesta on tilanne, jossa mikropalvelu ei pysty vastaamaan siihen saapuvaan kyselyyn normaalissa ajassa. Tämä aiheuttaa erityisiä haasteita synkroniseen kommunikointiin perustuvassa mikropalveluarkkitehtuurissa, jossa järjestelmään saapuvaan kyselyyn vastaamiseen saattaa osallistua useita mikropalveluita. Mikäli yksikin vastauksen muodostamiseen osallistuva mikropalvelu epäonnistuu, se hidastaa koko kyselyn toteutumista. Tavallista pidempään kestävät kyselyt voivat lopulta ruuhkauttaa koko järjestelmän käyttämällä kaiken palvelimella saatavilla olevan laskentakapasiteetin. Tällöin järjestelmä ei kykene ottamaan vastaan uusia kyselyjä laisinkaan, jonka vuoksi

ketjuuntuvien virhetilanteiden käsitteleminen on luotettavan mikropalveluarkkitehtuurin kannalta oleellista.

Mikropalveluarkkitehtuurissa vikasietoinen järjestelmä voidaan saavuttaa soveltamalla erilaisia suunnittelumalleja järjestelmään kuuluvien mikropalveluiden toteutuksessa ja tuotantoympäristöjen rakentamisessa. Mikropalveluarkkitehtuuriin liittyen virheiden käsittelyyn ja niistä palautumiseen on useissa lähteissä ehdotettu Nygardin (2007) esittelemiä katkaisija suunnittelumallia (Circuit Breaker) ja kriittisten palveluiden eristämistä (Bulkheads) (Newman, 2015, 212–214; Nadareishvili ym., 2016, 72; Francesco ym., 2017). Kriittiset palvelut voidaan eristää toimimaan omilla palvelimilla, jolloin järjestelmään kuuluvat muut mikropalvelut, eivät pääse vaikuttamaan niiden saavutettavuuteen. Tyypillisesti kuitenkin useita eri toimintoja toteuttavat mikropalvelut sijoitetaan samalle palvelimelle, jolloin katkaisija suunnittelumallin avulla pystytään suojautumaan ketjuuntuvilta virhetilanteilta. Montesin ja Weberin (2016) mukaan katkaisija-suunnittelumalli estää ketjuuntuvien virheiden tapahtumisen sulkemalla yhteyden liian monta virheellistä vastausta lähettäneeseen mikropalveluun. Yhteyden katkaiseminen voidaan toteuttaa eri kriteerien perusteella, kuten liian pitkän vastaukseen kuluneen ajan tai liian monen virheellisen vastauksen perusteella. Mikropalveluarkkitehtuurissa katkaisijat voidaan toteuttaa kyselyitä alemmalla tasolla toimiville mikropalveluille lähettävään rajapintaan, rajapinnan ja mikropalveluiden välille sijoitettuun välityspalvelimeen tai kyselyyn vastaavaan mikropalveluun (Montesi & Weber, 2016).

Katkaisijoiden toteuttamiseen on tarjoilla useita avoimeen lähdekoodiin perustuvia ohjelmistokirjastoja, kuten esimerkiksi Java-ohjelmointikielellä toteuttama Hystrix[1] tai C#-ohjelmointikielellä toteutettu Polly[2]. Näiden ohjelmointikirjastojen avulla voidaan toteuttaa asiakas- tai palvelinpuolella tapahtuva virheiden käsittely. Välityspalvelimen avulla toteuttavia ratkaisuja tarjoavat esimerkiksi HaProxy[3] ja NGINX[4]. Mikropalveluiden virheiden käsittelyssä on tärkeää myös huomioida, että järjestelmään kuuluvien muiden osien, kuten käyttöliittymien on pysyttävä käsittelemään osittain onnistuvia kyselyitä.

2.4 Mikropalveluarkkitehtuurin toteuttaminen

Mikropalveluarkkitehtuuriin perustuvien järjestelmien toteuttamiseen liittyy useita käytänteitä, joiden kautta voidaan tehostaa mikropalveluarkkitehtuuriin perustuvan järjestelmien rakentamista. Seuraavissa alaluvuissa käsitellään mitä erityispiirteitä mikropalveluarkkitehtuuriin perustuvan järjestelmän toteuttamiseen liittyy.

2.4.1 Mikropalveluiden suunnittelminen

Liiketoimintalogiikasta vastaavien mikropalveluiden kesken oikeanlaisen palvelujaon löytäminen on erittäin tärkeää, se vaikuttaa esimerkiksi

1 https://github.com/Netflix/Hystrix
2 https://github.com/App-vNext/Polly
3 https://www.haproxy.com
4 https://nginx.org

järjestelmän skaalautumiseen. Lisäksi huonosti toteutettu palvelujako johtaa tilanteeseen, jossa mikropalveluiden välinen verkon ylitse tapahtuva kommunikointi kasvaa, mikä huonontaa järjestelmän suorituskykyä. Oikean palvelujaon löytämistä kuvaillaan myös mikropalveluarkkitehtuurin yhdeksi keskeisimmistä haasteista (Taibi & Lenardruzzi, 2018; Soldani ym., 2018; Ghofrani & Lübke, 2018). Järjestelmän jakaminen mikropalveluiksi voidaan yleisellä tasolla aloittaa kartoittamalla järjestelmän vaatimuksiin kuuluvien liiketoimintaprosessien sisältöä. Pahl, Jamshidi ja Zimmermann (2018) ehdottavat liiketoimintaprosesseihin liittyvien verbien ja substantiivien löytämistä, jolloin esimerkiksi verkkokauppaan liittyvää liiketoimintaprosessia voitaisiin kuvailla sanoin: asiakas tilaa tuotteen verkkokaupasta. Näin ollen tilaus, tuote ja asiakas kuvaisivat liiketoimintaprosessiin liittyviä mikropalveluita.

Palvelujaon aikaansaamiseksi ehdotetaan kuitenkin useissa lähteissä käytettäväksi Evansin (2003) esittelemää järjestelmän liiketoimintavaatimuksia seuraamaan pyrkivää suunnittelumallia, josta käytetään yleisesti lyhennettä DDD (Domain Driven Design) (Millet & Tune, 2015. 178; Newman, 2015; Zimmermann, 2017; Jamshidi ym., 2018, Francesco ym, 2018; Grorani & Lubke, 2018). Vaikkakin DDD on lähtöisin ajalta ennen mikropalveluarkkitehtuuria, sen avulla voidaan helpottaa toimivan palvelujaon löytämistä. Millet ja Tune (2015, 14) kuvailevat DDD:tä ohjelmistokehittämisen lähestymistavaksi, jonka ydinajatus rakentuu liiketoimintalogiikan ja sen aiheuttamien riippuvuuksien löytämiseen ja ymmärtämiseen. DDD-suunnittelussa järjestelmän sisäisiä riippuvuuksia kuvataan aihealuemallin (Domain Model) avulla, jossa liiketoimintalogiikkaan kuuluvat osa-alueet kuvataan entiteetteinä (engl Domain Entities). DDD ei itsessään kuvaa miten arkkitehtuuri tulisi tarkalleen toteuttaa tai mitä arvoja aihealuemallin entiteettien tulisi sisältää, vaan pikemminkin sen avulla pyritään löytämään entiteettien muodostamat liiketoimintalogiikan kannalta oleelliset kokonaisuudet (Bounded Countext). Näitä kokonaisuuksia voidaan hyödyntää mikropalveluarkkitehtuuriin perustuvan järjestelmän palvelujakoa tehtäessä. Evansin (2003, 41) mukaan DDD-suunnittelumalli seuraa klassista ohjelmistoarkkitehtuuriin liittyvää ajatusmallia nimeltään huolenaiheiden erottaminen (Separation of Concerns), jonka mukaan samasta syystä muuttuvat järjestelmän osat tulee koota yhteen ja eri syistä muuttuvat järjestelmän osat tulee erottaa toisistaan (Evans, 2003, 41).

DDD:tä kohtaan on kuitenkin esitetty myös kritiikkiä, jonka mukaan siinä ei huomioida tarpeeksi järjestelmän laatuun liittyviä seikkoja. Gysel, Kölbener, Giersche ja Zimmermann, (2016) pyrkivät ratkaisemaan tämän ongelman ehdottamalla palveluiden jakamiseen algoritmi avusteista ratkaisua, joka pohjautuu ennalta määriteltyihin palvelujaon kannalta merkittäviin järjestelmän laatua kuvaaviin kriteereihin. Näiden pohjalta liiketoimintalogiikka voidaan jakaa mikropalveluiden toiminnollisuuksia kuvaaviin kokonaisuuksiin. Myös Rademacher, Sorgalla ja Sachweh (2018) pitävät DDD:tä liian suppeana kokonaisvaltaisen mikropalveluarkkitehtuurin suunnittelussa. He ehdottavat sen laajentamista kuvaamalla myös

aihealuemalliin liittyvien kokonaisuuksien välillä tapahtuvat operaatiot ja niiden parametrit. Toisena oleellisena muutoksena he ehdottavat mikropalveluarkkitehtuuria tukevien infrastruktuuripalveluiden lisäämisen osaksi aihealuemallia (Rademacher ym., 2018).

2.4.2 Mikropalveluiden kehittäminen

Namiotin ja Sneps-Sneppen (2014) mukaan mikropalveluiden itsenäisyys sekä verkon yli tapahtuva kommunikointi mahdollistavat mikropalveluarkkitehtuurin monikielisyyden. Näin ollen mikropalveluiden kehittämisessä voidaan hyödyntää ongelmaan parhaiten soveltuvaa ohjelmointikieltä (Namiot & Sneps-Sneppe, 2014). Myös Soldani ym. (2018) näkevät mikropalveluiden monikielisen kehittämisen tärkeänä ominaisuutena. Monikielisyyden lisäksi mikropalveluiden itsenäisyys mahdollistaa tehokkaan useiden palveluiden rinnakkaisen kehittämisen, jota pidetään myös yhtenä mikropalveluarkkitehtuurin suurena hyötynä (Dragoni ym., 2017).

Mikropalveluarkkitehtuuriin perustavaa järjestelmää rakennettaessa korostuu myös organisaation rakenne, jonka tulisi tukea mikropalveluiden kehittämistä mahdollisimman hyvin. Newman (2015, 191) ehdottaa mikropalveluarkkitehtuurin liittyvien haasteiden jakamista arkkitehtuuriin ja organisaation rakenteeseen liittyviin haasteisiin. Organisaation rakenteeseen liittyviin haasteisiin viitataan usein mikropalveluiden yhteydessä Conway'n lailla (1968), jonka mukaan järjestelmien arkkitehtuuri seuraa organisaation kommunikointi rakennetta. Näin ollen erillään toimivat sidosryhmät keskittyvät erityisesti ratkaisemaan itseensä vaikuttavia ongelmia. Toisin sanoen järjestelmän kehittämiseen ja ylläpitämiseen keskittyneet henkilöt eivät välttämättä pyri ratkaisemaan toisilleen oleellisia ongelmia. Tämä aiheuttaa erityisisä haasteita mikropalveluarkkitehtuurissa, sillä arkkitehtuurityyli esittelee suuren määrän monimutkaisuutta, jossa järjestelmään liittyvän kriittisen infrastruktuurin ja mikropalveluiden yhteistoiminnan tulisi olla saumatonta. Mikropalveluarkkitehtuuriin liittyen yleisesti pidetään hyvin merkityksellisenä DevOps-käytänteiden omaksumista, joiden avulla pystytään tehostamaan monimutkaisten järjestelmien kehittämistä. DevOpsin merkitystä mikropalveluarkkitehtuurissa käsitellään tarkemmin tutkielman kolmannessa luvussa.

2.4.3 Järjestelmien muuntaminen mikropalvelupohjaiseksi

Suunnittelussa tulisi aina kiinnittää erityistä huomiota liiketoimintavaatimuksiin, joiden pohjalta pystyttään analysoimaan, kuinka hyvin mikropalveluarkkitehtuuri soveltuu järjestelmän kehittämiseen. Vaikkakin mikropalveluarkkitehtuuri tarjoaa etuja helppoon skaalautuvuuteen ja suurten käyttäjämäärien tukemiseen. Nämä eivät kuitenkaan ole välttämättä vaatimuksia, joiden tarvitsee realisoitua heti ensimmäisen julkaisun jälkeen. Lisäksi monoliittiseen arkkitehtuuriin perustuvan järjestelmän rakentaminen on huomattavasti nopeampaa ja yksinkertaisempaa, koska palveluiden välistä

jakoa ei tarvitse miettiä. Tämän vuoksi Newman (2015, 249) ehdottaakin, että etenkin uusien järjestelmien suunnitteleminen olisi edelleen viisaampaa aloittaa rakentamalla monoliittinen järjestelmä, joka voidaan tarpeen vaatiessa jakaa mikropalveluiksi. Näin toimimalla pystytään ymmärtämään paremmin järjestelmän toiminnan kannalta kriittisiä kokonaisuuksia, joiden voidaan katsoa olevan tiukasti sidoksissa toisiinsa. Mikropalveluarkkitehtuurin yksi merkittävistä käyttökohteista on jo olemassa olevien järjestelmien muuntaminen mikropalveluiksi (Francesco, ym., 2018). Näin ollen myös mikropalveluarkkitehtuuria käsittelevässä tutkimuksessa on kiinnitetty huomiota, siihen kuinka olemassa olevia monoliittiseen arkkitehtuuriin perustuva järjestelmä voidaan muuntaa mikropalveluksi.

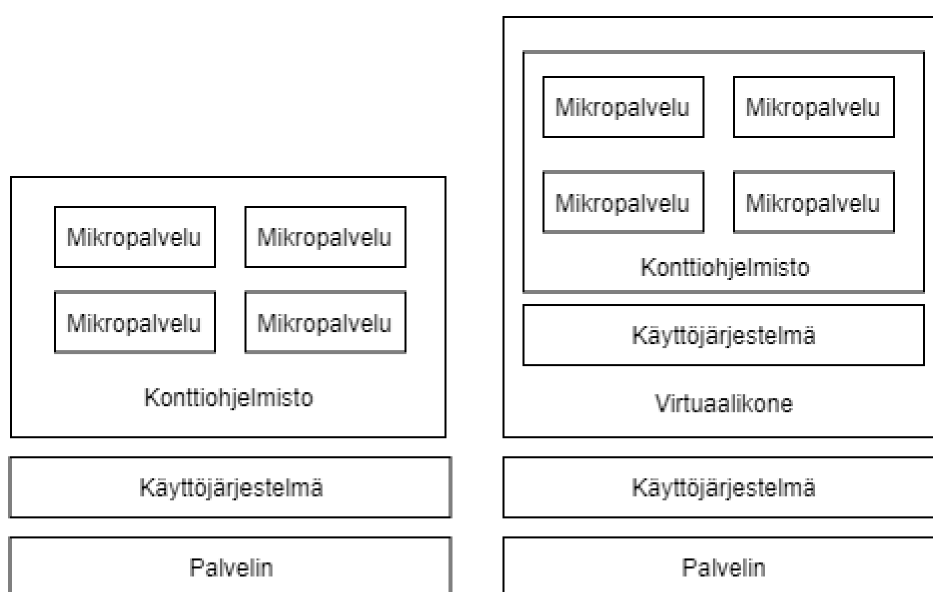
Balalaie ym. (2018) tunnistivat yhteensä 13 muunnosprosessiin hyödynnettävissä olevaa toimenpidettä. Samalla toimenpiteet kuvaavat myös hyvin mikropalveluarkkitehtuurille tyypillisiä elementtejä. Positiivisia käytännön kokemuksia suurenluokan monoliittisen järjestelmän muuntamisesta mikropalveluarkkitehtuuriin ovat esittäneet Buccihiarone ym. (2018), jotka muunsivat pankkisektorilla toimivan organisaation monoliittisen järjestelmän mikropalveluarkkitehtuuriin. Heidän mukaansa onnistuneen migraation avaintekijöitä olivat mikropalveluiksi jaettujen toiminnollisuuksien julkaisemisen automatisointi sekä järjestelmän sisäisen kommunikoinnin uudelleen organisointi. Tämä saavutettiin rakentamalla järjestelmään kuuluville mikropalveluille omat julkaisuputket, joiden rakentamiseta tukivat Docker-kontit ja Docker Swarm-konttienhallintaohjelma. Lisäksi järjestelmän sisäinen kommunikointi muutettiin viestijono pohjaiseksi, jossa järjestelmän kommunikointi perustui mikropalveluiden väliseen orkestrointiin (Buccihiarone ym., 2018)

2.4.4 Julkaiseminen

Mikropalveluarkkitehtuuriin on sisään rakennettu ajatus itsenäisten palveluiden nopeasta julkaisemisesta, joka perustuu suuren järjestelmän jakamiseen pieniin palveluihin. Useista mikropalveluista koostuvien järjestelmien manuaalinen hallinnointi on erittäin työlästä ja virhealtista, sillä järjestelmät sisältävät tyypillisesti myös infrastruktuuriin liittyviä elementtejä, jotka vaikuttavat palveluiden skaalautuvuuteen, monitorointiin ja löydettävyyteen (Jamshidi, ym., 2018). Mikropalveluarkkitehtuuriin perustuvat järjestelmät toimivat tyypillisesti pilvipohjaisissa palvelinympäristöissä ja niitä julkaistaan ja hallinnoidaan usein valmiita pilvialustoja (Platform as a Service) hyödyntäen (Pahl & Jamshidi, 2016; Francesco ym., 2017; Pahl, Jahmsihid & Zimmermann, 2018). Tunnettuja pilvialustoja ovat esimerkiksi suurten teknologiayritysten tarjoamat tuotteet, kuten Amazon Web Services, Microsoft Azure ja Google Cloud. Hallinnointia helpottavien pilvialustojen lisäksi mikropalveluiden julkaisemiseen hyödynnetään usein kolmansienosapuolien toteuttamia työkaluja. Näistä työkaluista merkittävimpiä ovat konttiteknologiat ja jatkuvan toimittamisen mahdollistavat julkaisunhallintatyökalut (Balalaie ym., 2016; O'Connor ym., 2017).

Mikropalveluarkkitehtuuriin perustuvien järjestelmien yhtenä tärkeimmistä mahdollistajista pidetään konttiteknologioita, joiden avulla itsenäisiä mikropalveluita voidaan helposti julkaista (Newman, 2015, 126; Balalaie ym., 2016; Kang, Le & Tao, 2016; Nadareishvili ym. 2016; Cerny ym., 2017). Konttiteknologiat perustuvat Linux-käyttöjärjestelmään sisäänrakennettuihin Linux-kontteihin. Ne mahdollistavat käyttöjärjestelmän tasolla tapahtuvan virtualisoinnin, eristämällä kontin sisällä ajettavan prosessin käyttöjärjestelmässä toimivista muista prosesseista (Merkel, 2014). Konttien hyödyntämiseen on tarjolla useita eri vaihtoehtoja, mutta suosituin tarjolla olevista ratkaisuista on Docker, joka toimii Mac OS, Windows- ja Linux-käyttöjärjestelmissä (Sysdig, 2018).

Dockerin (2018) toiminta perustuu Linux-konttien hyödyntämiseen. Dockerissa ideana on paketoida julkaistava sovellus kaikkine riippuvuuksineen näköistiedostoksi (Image), joista pystytään muodostamaan tosistaan erillään ajettavia kontteja (Container). Konttien välille muodostetaan Dockerin toimesta oma verkko, jonka avulla konteissa toimivat ohjelmat kommunikoivat keskenään sekä muun maailman kanssa. Myös eri palvelimilla toimivat Docker-instanssit voivat muodostaa verkkoja, joiden avulla pystytään keskitetysti hallinnoimaan eri palvelimille sijoitettuja kontteja (Docker, 2018). Docker eroaa perinteisestä virtualisoinnista siinä mielessä, että konteissa ajettavat ohjelmat jakavat palvelimelle asennetun käyttöjärjestelmän ytimen, kun taas perinteisessä virtualisoinnissa jokaisessa virtuaalikoneessa toimii oma käyttöjärjestelmä (Martin, Raponi, Combe & Di Pietro, 2018). Tämän vuoksi kontteihin perustuva virtualisointi (kuvio 4) on huomattavasti perinteistä virtualisointia kevyempi vaihtoehto, koska palvelimella toimivalle käyttöjärjestelmä ytimelle välitettävät kutsut kulkevat suoraan Dockerin tarjoaman rajapinnan kautta. Vaikkakin suorituskykyä pidetään yhtenä saavutettavista eduista, Bernsteinin (2014) mukaan on hyvin tyypillistä, että Dokcer asennetaan palvelimella ajettavaan virtuaalikoneeseen (kuvio 4).



KUVIO 4 Mikropalveluiden julkaiseminen konttien avulla.

Suurempana hyötynä voidaan nähdä helppo riippuvuuksien hallinta, sekä nopea ja helposti automatisoitavissa oleva ympäristöjen hallinta (Bernstein, 2014). Dockerin käyttöä mikropalveluarkkitehtuurin yhteydessä tutkineiden Kangin ym. (2016) mukaan etuna perinteiseen virtualisointiin ovat konttien helppo siirreltävyys sekä niiden yksinkertainen elinkaaren hallinta ja ympäristöjen yhdenmukaisuus. Dockerin avulla esimerkiksi testaus- ja tuotantoympäristöt on helppo pitää identtisinä, sillä kaikki riippuvuudet, kuten ohjelmiston ajamiseen tarvittavat kirjastot paketoidaan jokaiseen näköistiedostoon erikseen. Tämä myös mahdollistaa sen, että yhdellä palvelimella voidaan helposti julkaista eri ohjelmointikielillä toteutettuja järjestelmiä, ilman että, palvelimelle tarvitsee erikseen asentaa jokaiseen ohjelmointikieleen liittyvät riippuvuudet. (Kang, 2016). Docker (2018) sisältää myös julkisen näköistiedostorekisterin, jonka kautta käyttäjät pystyvät automaattisesti omia näköistiedostoja luodessaan lataamaan omaan sovellukseensa liittyvät riippuvuudet ja myös julkaisemaan omia näköistiedostojaan. Docker tarjoaa myös ominaisuuden ylläpitää suljettuja rekisterejä, joiden avulla organisaatiot voivat turvallisesti jakaa yksityisenä pidettäviä näköistiedostoja. (Docker, 2018).

Konttiteknologioiden ja erityisesti Dockerin käytöllä pystytään kehittämään helposti julkaistavissa olevia mikropalveluarkkitehtuuriin perustuvia järjestelmiä. Suurten konttipohjaisten järjestelmien hallinnointiin voidaan hyödyntää esimerkiksi Mesos[1], Kubernetes[2] ja Docker Swarn[3] ohjelmistoja, jotka ovat konttipohjaisten palvelinklusterien hallintaan tarkoitettuja työkaluja (Balalaie ym., 2018; Bucchiarone ym, 2018).

Käytännön kokemuksia Dockerin hyödyistä ovat esittäneet Balalaie ym. (2016), jotka sovelsivat Dockeria muuntaessaan lokaalisti ajetun järjestelmän pilvipohjaiseksi mikropalveluita hyödyntäväksi järjestelmäksi. Dockeria hyödynnettiin jatkuvan julkaisemisen mahdollistavan infrasturktuurin rakentamisessa. Lisäksi Dokcerin Compose- tominnon avulla testiympäristön pystyttäminen kyettiin suorittamaan yhdellä Docker-komennolla (Balalaie ym., 2016). Myös Jaramillo, Nguyen ja Smart (2016) esittelivät miten Dockerin avulla pystytään tehostamaan mikropalveluarkkitehtuuriin perustuvaa kehittämistä. Heidän mukaansa Docker tukee mikropalveluiden monikielisyyttä eristämällä konttien sisällä toimivat mikropalvelut muista samalla palvelimella ajettavista prosesseista.

2.5 Yhteenveto mikropalveluarkkitehtuurista

Tässä luvussa käytiin läpi mikropalveluarkkitehtuuriin liittyviä keskeisiä ominaisuuksia ja piirteitä, jotka tunnistettiin aiemman kirjallisuuden perusteella. Mikropalveluarkkitehtuuri jakaa useita piirteitä SOA-arkkitehtuurin kanssa, mutta niiden välillä on myös eroavaisuuksia. Merkittävimpinä voidaan pitää SOA-arkkitehtuurille tyypillistä tapaa pyrkiä jakamaan mahdollisimman paljon toiminnollisuuksia niin koodi kuin tietokantatasollakin. Mikropalveluarkkiteh-

1 http://mesos.apache.org
2 https://kubernetes.io
3 https://docs.docker.com/engine/swarm

tuurissa puolestaan tavoitteena on pyrkiä eriyttämään toimintoja mahdollisimman paljon siten, että mikropalveluiden itsenäisyys pystytään säilyttämään. Mikropalveluiden kehittämisessä korostuu erityisesti suunnittelun tärkeys sekä mikropalveluiden välisten rajapintojen toteutus. Mikropalveluiden skaalautuvuutta voidaan pitää yhtenä merkittävistä tekijöistä mikropalveluarkkitehtuuriin yleistymisessä. Skaalautuvuuteen liittyy oleellisesti järjestelmän jatkuva monitorointi ja mikropalveluiden automaattinen löydettävyys. Mikropalveluiden julkaisemiseen liittyen tärkeänä pidetään pilvipohjaisten julkaisualustojen hyödyntämistä sekä mikropalveluiden paketoimista kontteihin, joiden avulla voidaan tehostaa mikropalveluarkkitehtuuriin perustuvan järjestelmän julkaisemista ja hallinnointia.

Jamshidin ja Pahl (2016) mukaan mikropalveluarkkitehtuuria tulisi tarkistella kokonaisuutena, johon oleellisesti liitetään myös järjestelmää tukeva infrastruktuuri. Myös Richardson (2018, 94) korostaa infrastruktuurin merkitystä mikropalveluarkkitehtuuriin liittyen. Näin ollen on mielekästä tarkastella myös tämän työn osalta millaisien toimintamallien ja prosessien kautta pystytään tehostamaan mikropalveluarkkitehtuuriin perustuvaa kehittämistä. Seuraavassa luvussa esitellään, millaisilla prosesseilla ja toimintamalleilla voidaan tehostaa mikropalveluarkkitehtuuriin pohjautuvien järjestelmien kehittämistä ja miksi niiden merkitys korostuu juuri mikropalveluarkkitehtuurin yhteydessä.

3 MIKROPALVELUIDEN KEHITTÄMISTÄ TUKEVAT TOIMINTAMALLIT JA PROSESSIT

Mikropalveluarkkitehtuuriin perustuvan järjestelmän kehittämisessä on useita haasteita, joihin voidaan vastata omaksumalla kehitystyötä tukevia sekä helpottavia toimintamalleja. Mikropalveluarkkitehtuuriin pohjautuva järjestelmä saattaa sisältää useita satoja mikropalveluita, jotka on hajautettu useille eri palvelimille. Tällöin järjestelmän kehittäminen, testaus ja julkaiseminen muuttuu erittäin hankalaksi. Pienien itsenäisten palveluiden suuri määrä voidaan nähdä haasteena, mutta se toimii myös oleellisena mahdollistajana maksimaalista ketteryyttä tavoiteltaessa. Mikropalveluarkkitehtuuriin perustuvaa kehittämistä tukee erityisesti Beckin (1999) esittelemästä XP-kehitysmenetelmästä (Extreme Programming) lähtöisin oleva jatkuva integraatio (Continuous Integration) ja tästä jalostunut jatkuvan toimittamisen (Continuous Delivery) periaate (Humble & Farley, 2011).

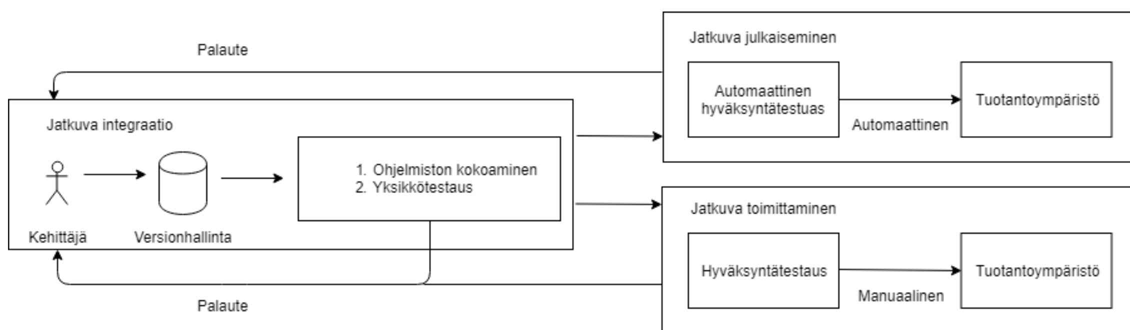
Jatkuvuuteen perustuvien julkaisumenetelmien hyödyntäminen on tunnistettu mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä tehostaviksi menetelmiksi (Dragoni, ym., 2017). Jatkuvan integraation ja toimittamisen saavuttaminen on kuitenkin haastava ja aikaa vievä prosessi. Tätä tarvetta täyttämään on noussut ohjelmistojen kehittämistä tukeva DevOps-toimintamalli. Moleskyn ja Humblen (2011) mukaan termi DevOps muodostuu englanninkielisistä sanoista Development ja Operations. Toimintamallin tarkoituksena on saattaa yhteen ohjelmistotuotannon eri osa-alueista vastaavat henkilöt ja muodostaa DevOps-tiimejä, jotka kykenevät tehokkaasti tukemaan monimutkaistuvien ja kasvavien järjestelmien kehittämistä, testaamisesta, julkaisemista ja ylläpitoa (Molesky & Humble, 2011).

Viime vuosina DevOpsin suosio on kasvanut selkeästi. Tähän varmasti vaikuttaa omalta osaltaan myös mikropalveluarkkitehtuurin yleistyminen, joka vaatii toimiakseen monimutkaisten ympäristöjen rakentamista (Puppet & Splunk, 2018). Ohjelmistokehittämiseen hyödynnettäviä menetelmiä kartoittavissa tutkimuksissa on tunnistettu, että tutkimusintressi on siirtymässä perinteisistä ketteristä viitekehyksistä kohti jatkuvaa julkaisemista ja DevOps-toimintamallia käsitteleviin teemoihin (Dingsøyr & Lassenius, 2016; Shahin,

Babar & Zhu, 2017). Myös Zimmermannin (2017) mukaan perinteisten ketterien menetelmien sijaan, kasvavaa kiinnostusta on havaittavissa DevOpsia ja jatkuvaa toimittamista kohtaan. DevOps ja jatkuva toimittaminen pyrkivät ratkaisemaan ongelmia, joita mikropalveluarkkitehtuuri asettaa, korostamalla automaation ja yhteistyön tarvetta monimutkaisia järjestelmiä kehitettäessä. Toisaalta myös mikropalveluarkkitehtuurin modulaarisuutta pidetään yhtenä tärkeänä mahdollistajana jatkuvaa toimittamista tukevaa infrastruktuuria rakennettaessa (Callanan & Splilane, 2016). Myös monissa mikropalveluarkkitehtuuria käsittelevissä julkaisuissa mainitaan, kuinka DevOps toimii tärkeänä mahdollistajana mikropalveluarkkitehtuurin hyödyntämisessä (Kang ym., 2016; Balalaie ym., 2016; Francesco ym., 2017; Jamshidi ym., 2018). Seuraavissa alaluvuissa käsitellään tarkemmin, järjestelmien julkaisemissa hyödynnettäviä prosesseja, DevOpsiin liittyviä periaatteita ja miten nämä mahdollistavat mikropalveluarkkitehtuurin tehokkaan hyödyntämisen.

3.1 Julkaisemisen hallintaprosessit

Ohjelmistojen julkaisemisen hallinta (Release Engineering) on ohjelmistokehittämisen osa-alue, jonka tavoitteena on tuottaa mahdollisimman tehokkaita prosesseja kehitettävien järjestelmien testaamiseen ja asiakkaille toimittamiseen. (Adams & McIntosh, 2016). Järjestelmien julkaisemiseen liittyvät toimenpiteet voidaan jakaa kolmeen prosessiin (kuvio 5). Julkaisuprosessin ensimmäinen vaihe on nimeltään jatkuva integraatio, jota voidaan pitää edellytyksenä sitä seuraavalle jatkuvalle toimittamiselle tai jatkuvalle julkaisemiselle (Shahin ym., 2017).



KUVIO 5 Julkaisemisen eri vaiheet (Shahin ym., 2017).

Laukkanen, Itkonen ja Lassenius (2017) huomauttavat, että termejä kuitenkin käytetään myös tieteellisessä tutkimuksessa huolimattomasti. Esimerkiksi joissain tapauksissa puhutaan jatkuvasta julkaisemista, vaikka todellisuudessa käsitellään jatkuvaa toimittamista. Osaltaan termien epäselvyyteen vaikuttaa se, että organisaatiot soveltavat julkaisemiseen liittyviä prosesseja kukin tavallaan, joissain tapauksissa julkaisemiseen liittyvät toimenpiteet saattavat sisältää useita manuaalisia työvaiheita, kun taas toisaalla julkaisuprosessi voi olla automati-

soitu kokonaan (Ståhl & Bosch, 2014). Täten on tärkeää tehdä selkeä ero termien välille ja perustella, mitä mikäkin termi tarkoittaa.

3.1.1 Jatkuva integraatio

Humble ja Farley (2011, 109–112) kuvailevat jatkuvaan integraatioon liittyviä toimenpiteitä seuraavasti. Jatkuvalla integraatiolla tarkoitetaan mahdollisuutta julkaista järjestelmään tehdyt muutokset testiympäristöön, jossa ohjelmakoodi rakennetaan ja testataan. Täten kehitystiimi saa viimeisimmistä muutoksista välittömän palautteen. Mikäli testiympäristöön julkaistu versio ei läpäise testejä voidaan havaitut virheet korjata heti. Tämän toimintaperiaatteen avulla saadaan aikaan nopea palautesykli. Nopean ja jatkuvan palautteen ansiosta järjestelmä kyetään pitämään toimintakuntoisena mahdollisimman tehokkaasti. Jatkuvan integraation toteuttamisessa on tärkeää, että kehitystiimi pyrkii lisäämään tekemänsä muutokset versionhallintaan mahdollisimman usein, tällöin integraatio prosessista saadaan sulava ja versionhallinnassa oleva ohjelmakoodi kuvastaa koko järjestelmän sen hetkistä tilaa. Jatkuvaan integraatioon liittyy oleellisesti myös läpinäkyvyyden korostaminen, jonka vuoksi oleellisilla sidosryhmillä on mahdollisuus tarkastella viimeisimpiä järjestelmään tehtyä muutoksia sekä automaattisesti suoritettujen testien tuloksia (Humble & Farley, 2011, 109–112).

Forsgren, Humble ja Kim (2018) esittävät samankaltaisen kuvauksen jatkuvan integraation mahdollistavista tekijöistä. Heidän mukaansa jatkuva integraatio edellyttää, että kaikkien kehitystiimin jäsenten on pyrittävä tekemään kehitystyönsä versiohallintajärjestelmän saamaan haaraan, jolloin kyseinen haara indikoi kehitettävän järjestelmän viimeisintä tilaa. Jotta tämä olisi mahdollista, on järjestelmän arkkitehtuurin oltava helposti laajennettavissa. Tämä taas mahdollistaa suurten toiminnallisuuksien pilkkomisen pieniin osiin, siten että järjestelmä säilyy koko kehitysvaiheen ajan toimintakuntoisena. Lisäksi järjestelmään tehdyistä muutoksista saatavan palautteen on oltava automaattista, jatkuvaa ja selkeää (Forsgren ym., 2018).

Jatkuva integraatio ei kuitenkaan aina toteudu odotetulla tavalla. Ståhl, Mårtensson ja Bosch (2017) tutkivat, miten organisaation koko vaikuttaa jatkuvaan integraation toteuttamiseen. Tutkimuksessa selvisi, että yksittäisen kehittäjän näkökulmasta etenkin verrattain suurissa ohjelmistoprojekteissa jatkuva integraatio toteutui huomattavasti hitaammin kuin pienemmissä projekteissa. Tulokset ovat erityisesti ristiriidassa jatkuvaan integraatioon liittyvän perusajatuksen suhteen, jonka mukaan kehittäjien tulisi pyrkiä integroimaan tekemänsä muutokset versionhallintaan mahdollisimman usein. Tästä voidaan päätellä, että jatkuvan integraation saavuttaminen muuttuu haastavammaksi, mitä suuremmiksi järjestelmät kasvavat. Ratkaisuna Ståhl ym. (2017) ehdottavat kehitettävän järjestelmän arkkitehtuurin modulaarista suunnittelua, joka mahdollistaa suurien ominaisuuksien jakamisen pieniin kokonaisuuksiin.

3.1.2 Jatkuva toimittaminen

Jatkuva toimittaminen kuvaa prosessia, jonka mukaan kehitettävä järjestelmä pyritään pitämään tilassa, jossa se voidaan julkaista tuotantoympäristöön, milloin vain. Jatkuvaa integraatiota seuraa tyypillisesti, jatkuvan toimittamisen prosessi, joka lisää jatkuvaan integraatioon yhden uuden askeleen, jossa koottu ja yksikkötestauksen läpäissyt ohjelmisto työnnetään testiympäristöön, jossa voidaan suorittaa käyttöliittymään tai muihin järjestelmiin yhteydessä olevia integraatio testejä (Humble & Farley, 2011, 327). Odotettavasti jatkuvaan toimittamiseen ja integraatioon liittyvät haasteet ovat hyvin samankaltaisia. Laukkanen ym. (2017) suorittivat katsauksen jatkuvaan integraatioon ja jatkuvaan toimittamiseen liittyviin haasteisiin. He tunnistivat yhteensä kuusi teemaa, jotka olivat ohjelmakoodin rakentaminen, järjestelmän suunnittelu, testaus, julkaiseminen ja organisaation rakenne. Heidän mukaansa suurimmat haasteet liittyivät järjestelmien testaamiseen, sillä luotettavan jatkuvan julkaisemisen edellytyksenä ovat kattavat testitapaukset, joiden avulla voidaan varmistaa järjestelmän uuden version toimivuus. Erityisiä haasteita järjestelmien testauksessa aiheuttivat epäluotettavat testitapaukset, jotka tuottivat satunnaisesti vaihtelevia tuloksia. Myös huonot järjestelmän arkkitehtuuriin liittyvät valinnat aiheuttivat tilanteita, joissa oleellisia toiminnallisuuksia oli vaikea tai mahdoton testata. Lisäksi liian kauan kestäviä testitapauksia pidettiin ongelmallisina, koska ne hidastivat palautesykliä.

3.1.3 Jatkuva julkaiseminen

Jatkuva julkaisemien on varsin uusi toimintatapa joka, voidaan nähdä jatkuvan integraation laajennuksena, joka realisoi jatkuvaa toimittamista tukevan ajatuksen. Jatkuvassa julkaisemisessa tarkoituksena on julkaista jokainen muutos suoraan tuotantoympäristöön (Claps ym, 2015). Täten voidaan todeta, että jatkuva integraatio on edellytys jatkuvalle julkaisemiselle. Jatkuvan julkaisemisen saavuttaminen on erittäin haastavaa eikä se sovellu kaikkiin tilanteisiin. On hyvin tyypillistä, että tuotantoon tapahtuva julkaisuprosessi sisältää käyttäjätestaus vaiheen, joka on usein vaikeasti automatisoitavissa (Humble & Farley, 2011, 266–267).

Erityisiä haasteita jatkuvaan julkaisemiseen liittyen on löydetty esimerkiksi sulautettuihin järjestelmiin keskittyvässä ohjelmistokehittämisessä, jossa järjestelmien testaaminen aiheuttaa erityisiä haasteita, koska tuotantoympäristöjen kaltaisten automatisoitujen testiympäristöjen toteuttaminen on usein vaikeaa (Rodríguez, ym., 2017). Jatkuva julkaiseminen asettaa myös muita haasteita, esimerkiksi vaikka julkaisuprosessi sisältäisi useita automaattisia testausvaiheita on suoraan tuotantoon tapahtuva julkaiseminen aina riskialtista, sillä järjestelmän testaajina toimivat tällöin loppukäyttäjät. Tämän vuoksi automaattisten testitapauksien suunnittelemiseen ja toteuttamiseen on kiinnitettävä erityistä huomiota. Esimerkiksi testitapauksissa käytettävän datan on muistutettava muodoltaan tuotannossa olevaa dataa, lisäksi testiympäristön on vastattava

tarkkaan tuotantoympäristöä (Claps ym., 2015). Monissa kriittisissä järjestelmissä myös tuotantoon tapahtuvat päivitykset on kyettävä suorittamaan niin, että järjestelmä säilyy toimintakuntoisena päivityksen ajan. Suoraan tuotantoympäristöihin tapahtuvissa julkaisuissa voidaan hyödyntää erilaisia julkaisustrategioita, joiden avulla pystytään varmistumaan uuden version toimivuudesta ja toisaalta palautumaan nopeasti mahdollisista virhetilanteista. Esimerkiksi vanhaa ja uutta järjestelmäversioita voidaan ajaa rinnakkain, siten että uuteen versioon kohdistuvaa kuormaa kasvatetaan asteittain ja mikäli virheitä ilmenee, voidaan käyttäjät ohjata takaisin vanhaan versioon (Adams & McIntosh, 2016).

Kuten voidaan havaita jatkuvan julkaiseminen saavuttaminen ei ole yksinkertainen prosessi ja se vaatii paljon suunnittelua ja työtä. Clapsin ym. (2015) suorittaman tapaustutkimuksen mukaan jatkuvaan julkaisemiseen liittyvät haasteet voidaan jakaa sosiaalisiin ja teknisiin haasteisiin. Keskeisimpinä teknisinä haasteina pidettiin jatkuvaa julkaisemista tukevan infrastruktuurin rakentamista, kehitystyön jakamista tarpeeksi pieniin osiin ja järjestelmän toimintakuntoisena pitämistä päivitysten aikana. Lisäksi testitapauksista saatu ristiriitainen palaute aiheutti ongelmia, koska kehitystiimi ei voinut olla varma uuden kehitysversion soveltuvuudesta tuotantoon. Keskeisimpiä sosiaalisia haasteita olivat, jatkuvan toimittamisen aiheuttama kehitystiimiin kohdistuva paine, julkaisuprosessin monimutkaisuus erityisesti kokemattomille kehittäjille, organisaation sitoutuminen jatkuvan julkaisemisen periaatteisiin ja jatkuvan julkaisemisen johdosta saatavan jatkuvan palautteen analysointi (Claps ym., 2015).

3.2 DevOps

DevOps-toimintamallin avulla pystytään tehostamaan järjestelmien kehittämistä, julkaisua ja hallinnointia (Ebert, Gallardo, Hernantes & Serrano, 2016; Zhu, Bass & Champlin-Scharff, 2016). Lwakataren, Kuvajan ja Oivon (2015) mukaan DevOpsin onnistunut hyödyntäminen edellyttää jonkin ketterän viitekehyksen käyttöä. Lisäksi DevOps-toimintamallin omaksumista pidettiin edellytyksenä jatkuvan toimittamisen saavuttamisessa ja sen katsottiin seuraavan Lean-ajattelusta lähtöisin olevia periaatteita. Termiä DevOps pidetään edelleen hyvin häilyvänä, vaikka aiheeseen liittyvän tieteellisen tutkimuksen määrä on selkeässä kasvussa. Tämä voidaan havaita tarkastelemalla viime vuosina julkaistuja merkittäviä kartoitustutkimuksia, joissa kuvaillaan miten DevOps käyntänteet ovat yleistyneet (Jabbari, Ali, Petersen & Tanveer, 2016; Jabbari, Nauman, Petersen ja Tanveer, 2018). Dyckin, Pennersin ja Lichterin (2015) mukaan DevOps-toimintamallin tavoitteena on korostaa kehitystiimin ja järjestelmän ylläpidosta ja hallinnoinnista vastaavien henkilöiden yhteistyötä. Yhteistyön avulla pyritään toimittamaan parempilaatuisia järjestelmiä, vastaamaan nopeasti muuttuneisiin vaatimuksiin ja ratkaisemaan tuotantoympäristöissä ilmenneitä ongelmia. Lisäksi DevOpsissa korostetaan automaation tärkeyttä, jota hyödyntämällä voidaan tehostaa jatkuvuuteen perustuvien julkaisuprosessien raken-

tamista, ympäristöjen pystyttämistä sekä monitoroimaan ja mittamaan järjestelmän suorituskykyä (Dyck ym., 2015; Lwakatare ym 2015).

DevOpsiin liittyviä toimintaperiaatteita on pyritty määrittelemään useiden eri tahojen toimesta. Smedsin, Nybomin ja Porresin (2015) mukaan DevOps voidaan jakaa kahteen kategoriaan, joiden avulla toimintamalliin liittyviä periaatteita pystytään analysoimaan tarkemmin. Heidän mukaansa DevOps koostuu kyvyistä ja mahdollistajista. Kyvyt sisältävät DevOpsin kannalta oleellisia toimintatapoja ja periaatteita, joiden avulla pystytään tehostamaan ohjelmistokehittämiseen liittyviä prosesseja. Mahdollistajat ovat puolestaan organisaation kulttuuriin ja teknologisiin ratkaisuihin liittyviä muutoksia, joiden avulla kyvyt voidaan realisoida. DevOpsin kannalta oleellisia teknisiä kykyjä ovat jatkuva suunnittelu, testaaminen, monitorointi, julkaiseminen ja nopea virheistä palautuminen. Näiden teknisten kykyjen mahdollistajina organisaation kulttuurin näkökulmasta toimivat korostunut yhteistyö, kommunikaatio, tarkkaan määritellyt tavoitteet ja vastuut sekä oppimista ja uuden kokeilemista tukeva ilmapiiri. Teknologisina mahdollistajina tunnistettiin erityisesti valmiiden työkalujen hyödyntäminen, joiden avulla voidaan automatisoida prosesseja (Smeds ym., 2015). Samankaltaisiin tuloksiin päätyivät myös Lwakatare ym. (2015) määritellessään DevOpsin keskeisiä ulottuvuuksia. Heidän mukaansa näitä olivat yhteistyö, automaatio, monitorointi ja mittaaminen. Viimeisimmän kattavan DevOpsia määrittelemään pyrkivän kartoitustutkimuksen mukaan DevOpsin keskeiset periaatteet rakentuvat yhteistyön ja automaation ympärille (taulukko 1). Lisäksi tutkimuksessa tunnistettiin yhteensä 29 ohjelmistokehitykseen liittyvää toimintatapaa, joiden voidaan katsoa olevan osa DevOps käytänteitä (Jabbar ym., 2018).

TAULUKKO 1 DevOpsiin liittyvät periaatteet (Jabbar ym., 2018).

Periaate	**Kuvaus**
Tiedon jakaminen	Korostuneen yhteistyön tavoitteena on jakaa mahdollisimman paljon tietoa, jotta DevOpsi tiimin jäsenet kykenevät vastamaan järjestelmän kehittämisen eri osa-alueista.
Automaatio	Automaation avulla minimoidaan kehittämiseen, testaamiseen ja julkaisemiseen liittyviä työvaiheita
Jaettu vastuu	DevOps-tiimi vastaa yhdessä järjestelmän kehittämisestä, testaamisesta ja ylläpidosta.
Jatkuvuus	Jatkuva toimittaminen, jatkuva integraatio ja jatkuva seuranta
Monitorointi	Järjestelmien jatkuva ja automatisoitu monitorointi kehityksen eri vaiheissa sekä tuotantoympäristöissä
Koostettavuus	DevOps voidaan jakaa elementteihin, jotka määrittelevät toimintamalliin liittyvät keskeiset tekijät.

Jabbarin ym. (2018) tunnistamista DevOpsin keskeisistä periaatteista voidaan havaita, että niiden avulla pyritään ratkaisemaan Laukkasen ym. (2017) esiin tuomia järjestelmien julkaisuun ja organisaation rakenteeseen liittyviä haasteita. Toisaalta DevOpsin nimissä suoritettevista 29 tunnistetusta toimenpiteestä voidaan myös todeta, että DevOps ei ole tarkkaan rajattu viitekehys, jota myös korostetaan useissa muissa tutkimuksissa (Smeds ym., 2015; Dyck ym., 2015; Lwakatare ym., 2015). Yllä esitellyistä määritelmistä voidaan kuitenkin nostaa esiin selkeästi toistuvia teemoja, joiden ympärillä DevOpsin nimissä suoritettavat toimenpiteet tapahtuvat. Seuraavissa alaluvuissa perehdytään tarkemmin DevOpsiin liittyviin periaatteisiin.

3.2.1 Organisaatio ja kulttuuri

Humble ja Molensky (2011) kuvailevat perinteistä ohjelmiston julkaisuprosessia kehittäjien ja ylläpitäjien näkökulmasta seuraavasti. Yleensä kehitystiimi on vastuussa järjestelmän kehittämisestä ja testaamisesta, jonka jälkeen järjestelmä luovutetaan operaatioista vastaavalle tiimille. Kyseinen toimintamalli asettaa useita haasteita esimerkiksi tilanteissa, joissa päivitysprosessi perustuu kehitystiimin tuottamaan dokumentaatioon, jonka avulla tapahtuva kommunikointi johtaa usein väärinymmärryksiin, mitkä aiheuttavat epäonnistuneita julkaisuja. Kahden erillisen tiimin tavoitteet ovat myös selkeästi ristiriidassa keskenään. Järjestelmän kehittäjät eivät välttämättä ole täysin tietoisia, miten ja millaisessa ympäristössä järjestelmää käytetään sen lopullisessa tuotantoympäristössä. Lisäksi kehitystiimin tavoitteena on usein ketteriä periaatteita seuraten julkaista mahdollisimman nopeasti uusia versioita, kun taas ylläpito pyrkii mahdollisimman toimintavarmaan ja vakaaseen järjestelmään. Uusien ominaisuuksien kehitys tyypillisesti luo tilanteita, joissa järjestelmään tehdyt muutokset vaikuttavat myös tuotantoympäristöihin. Järjestelmien ylläpidosta vastaavien henkilöiden tavoitteena on puolestaan mahdollisimman vakaa järjestelmä, jolloin epäselvät sekä useita manuaalisia vaiheita sisältävät julkaisuprosessit aiheuttavat ristiriidan kahden erillisen tiimin välille (Molesky & Humble, 2011).

Hüttermannin mukaan (2012, 25) järjestelmien julkaisemiseen liittyvät haasteet pyritään DevOpsissa ratkaisemaan yhdistämällä järjestelmän kehitys, laadunhallinta ja ylläpito yhdeksi yhtenäiseksi tiimiksi. Moniosaajista koostuva tiimi sisältää henkilöitä, jotka pystyvät suunnittelemaan, kehittämään ja testaamaan ja ylläpitämään järjestelmää, mutta toisaalta myös ymmärtävät miten järjestelmän tulisi käyttäytyä eri ympäristöissä. Lisäksi yhteistyön tavoitteena on helpottaa järjestelmien julkaisuun ja monitorointiin tarkoitettujen ratkaisujen käyttöönottoa (Hüttermann, 2012, 25).

Perinteisten roolien sekoittuminen asettaa kuitenkin haasteita uusien toimintamallien omaksumisessa, sillä tyypillisesti erillään toimineilla sidosryhmillä on usein ajan kuluessa muodostunut oma kulttuuri, jonka vuoksi teknologisten haasteiden lisäksi DevOpsissa on huomioitava, miten eri toimintakulttuureihin tottuneet sidosryhmät saadaan työskentelemään tehokkaasti yhteisten

tavoitteiden saavuttamiseksi. Riungu-Kalliosaaren, Mäkisen, Lwakataren, Tiihosen ja Männistön (2016) DevOpsiin liittyviä haasteita käsittelevässä tutkimuksessa yhdeksi keskeiseksi haasteeksi tunnistettiin toimintakulttuuriin liittyvä muutos, sitä myös pidettiin edellytyksenä DevOpsin onnistuneessa hyödyntämisessä. Erich, Armit ja Daneva (2017) totesivat myös tutkimuksessaan kehittäjien kokeneen haastavaksi uusien ominaisuuksien kehittämisen ja samaan aikaan tuotantoympäristöistä vastuussa olemisen. Lisäksi järjestelmien hallinnoinnista vastaavien henkilöiden ja kehittäjien erilaiset toimintatavat koettiin haastavana. Esimerkiksi ohjelmistokehittäjät kokivat ylläpitoon liittyvät tehtävät haastavina, koska niitä ei ollut tarkkaan määritelty ja ne toisaalta sisälsivät paljon erityistapauksia, jotka liittyvät ainoastaan tiettyihin järjestelmiin ja ympäristöihin.

3.2.2 Automaatio

Kuten DevOpsin määritelmistä voidaan havaita, yksi siihen oleellisesti liittyvistä termeistä on automaatio. Automaation merkitys muuttuu sitä tärkeämmäksi mitä suuremmiksi ja monimutkaisemmiksi järjestelmät kasvavat. Esimerkiksi massiivisia järjestelmiä kehittävät ja ylläpitävät yritykset, kuten Facebook hyödyntävät DevOps-käytänteitä (Feitelson, Frachtenberg & Beck, 2013). DevOpsissa tyypillisesti pyritään hyödyntämään mahdollisimman paljon valmiita työkaluja, joiden avulla voidaan saavuttaa korkeantason automaatio. Ebertin ym. (2016) mukaan DevOps-tiimin tulisi ensisijaisesti keskittyä kehittämään järjestelmää, ei rakentamaan työkaluja. Työkalujen avulla voidaan erityisesti tukea järjestelmien julkaisua, testaamista, monitorointia ja mittaamista. Jatkuvan integraation ja jatkuvan toimittamisen toteuttamiseen on tarjolla useita maksullisia sekä avoimeen lähdekoodiin perustuvia ratkaisuja, joiden avulla pystytään rakentamaan tehokkaita tapoja ohjelmistojen julkaisemiseen. Tyypillisesti julkaisusta on vastuussa omalle palvelimelle asennettava ohjelmisto, joka määritetään tarkkailemaan versionhallintaan lisättäviä muutoksia. Uusia muutoksia havaitessaan ohjelmisto noutaa versionhallintajärjestelmästä viimeisimmät muutokset sisältävän ohjelmakoodin ja lähettää sen palvelimelle, jossa ohjelmisto automaattisesti rakentaa koodin ja ajaa ennalta määritetyt testitapaukset ja työntää uuden ohjelmistoversion haluttuun palvelinympäristöön (Humble & Farley, 2011, 63).

Toinen suureen suosioon noussut työkalu on järjestelmien julkaisua helpottavien konttiteknologioiden hyödyntäminen, joiden avulla palvelin ja siinä ajattevat ohjelmistot pystytään eristämään toisistaan. Konttiteknologioiden käytössä on kuitenkin myös haasteensa, sillä mitä suuremiksi järjestelmät kasvavat, sitä monimutkaisemman palvelinarkkitehtuurin ne vaativat toimiakseen. Monimutkaisten konttiarkkitehtuurien hallintaan on luotu orkestrointityökaluja, joiden avulla pystytään hallinnoinmaan suuria järjestelmiä, jotka vaativat toimiakseen useita palvelimia. DevOps-työkalujen hyödyntämisestä on tehty myös tapaustutkimuksia, joissa esitellään miten organisaatiot ovat pystyneet tehostamaan järjestelmiensä julkaisua ja hallinnointia. Balalaie ym. (2016)

suorittivat tapaustutkimuksen, jossa organisaation tavoitteena oli muuntaa vanha monoliittinen järjestelmä mikropalveluarkkitehtuuri muotoon ja ottaa samalla käyttöön DevOps. Projektin tärkeänä teknologisena mahdollistajana toimivat useat jatkuvan toimittamisen mahdollistavat työkalut, joita hyödynnettiin infrastruktuurin rakentamiseen. Täten jokaiselle järjestelmään kuuluvalle mikropalvelulle kyettiin rakentamaan oma julkaisuputki, joka mahdollisti yksittäisten palveluiden ketterän ja riippumattoman kehittämisen. Giovanni, Brunner, Blöchlinger, Dudouet & Edmonds (2015) tutkivat miten automaation mahdollistavien työkalujen avulla voidaan luoda skaalautuvia järjestelmiä, joiden käsittelykapasiteettiä pystytään muokkaamaan reaaliaikaisesti järjestelmään kohdistuvan kuorman kasvaessa. Esimerkiksi tilanteissa, joissa järjestelmän toiminnan kannalta kriittinen palvelu ei vastaa, voidaan kyseisestä palvelusta luoda uusi ilmentymä toiselle palvelimelle. Ehdotettu ratkaisu rakentui etcd-agenttien avulla ylläpidetyn hajautetun tietovaraston ympärille, johon jokainen järjestelmään kuulunut mikropalvelu välitti loki- ja suorituskykyyn liittyviä tietoja. Tietojen avulla voitiin analysoida palveluihin kohdistuvaa kuormaa ja luoda uusia ilmentymiä järjestelmään kuuluvista mikropalveluista (Giovanni ym., 2015). Tehokkaiden automaattisten julkaisukäytänteiden, monitoroinnin ja mittaamisen tärkeys korostuu erityisesti, kun kyseessä hajautettuun mikropalveluarkkitehtuuriin perustuva järjestelmä, jonka palvelut toimivat useissa ympäristöissä ja voivat olla toteutettu eri teknologioilla.

3.3 Tutkimuksia DevOpsin hyödyntämisestä

DevOpsin käyttöönotosta ja soveltamisesta on tehty useita tapaustutkimuksia, joista käy ilmi, että toimintamallilla on selkeästi positiivisia vaikutuksia ohjelmistokehittämiseen liittyvien prosessien tehostamisessa. Elberzhagerin, Arifin, Naabin, Süßin ja Kobanin (2017) mukaan Devopsin käyttöönottoa suunnittelevien organisaatioiden tulisi pystyä vastamaan seuraaviin kysymyksiin:

1. Mitä organisaatio tavoittelee DevOps-toimintamallin käyttöönotolla.
2. Miten DevOps-toimintamalli esitellään organisaation sisällä.
3. Miten vastuut tulevat jakautumaan.
4. Mitä vaikutuksia toimintamallilla on järjestelmien arkkitehtuuriin ja ympäristöihin.

Elberzhager ym. (2017) sovelsivat yllä esitettyjä kysymyksiään tutkiessaan, kuinka teknologiayhtiö Fujitsu hyödynsi DevOpsia suurenluokan ohjelmistoprojektin yhteydessä. Projekti aloitettiin määrittelemällä tavoitteet. Ensimmäisenä tavoitteena oli analysoida DevOpsia, analyysin avulla pyrittiin luomaan oleellisten sidosryhmien kesken yhtenevä käsitys, siitä mitä käytänteitä DevOps pitää sisällään ja miten sitä oli aiemmin hyödynnetty vastaavanlaisissa projekteissa. Toisena tavoitteena oli ottaa käyttöön analyysin avulla löydetyt parhaat toimintamallit ja periaatteet. Kolmantena tavoitteena oli luoda projektiin osallis-

tuvista henkilöistä Fujitsun sisäisiä DevOps-asiantuntijoita, jotka voisivat tulevaisuudessa jakaa tietoaan muiden osastojen kanssa. Neljäntenä tavoitteena oli suunnitella ja toteuttaa määrälliset mittausmenetelmät, joiden avulla voitiin seurata DevOpsin käyttöönoton vaikutuksia. DevOps esiteltiin organisaation sisällä valitsemalla yksi projekti, johon sovellettiin DevOpsia hyödyntäen analyysin avulla saatuja tuloksia. Projektitiimi koostui ohjelmistokehittäjistä, käyttöliittymäsuunnittelijoista ja yhdestä järjestelmien ylläpidon asiantuntijasta. Projektin alusta asti tiimi työskenteli yhdessä jakaen tehtävät, joiden tavoitteena oli saavuttaa jatkuvaa julkaisemista tukeva infrastruktuuri. Onnistuneen käyttöönoton jälkeen toimintamallin esittelemistä jatkettiin muihin kehitystiimeihin. DevOpsin käyttöönoton tuloksena projektissa aloitettiin hyödyntämään jatkuvaa julkaisemista tukevaa prosessia, jonka ansiosta uusia ohjelmistoversioita pystyttiin julkaisemaan tehokkaammin ja toisaalta vastaamaan muuttuneisiin vaatimuksiin nopeammin. Tutkimuksessa myös huomautettiin, että DevOpsin käyttöön ottamista helpotti erityisesti organisaatiossa jo käytössä olleet ketterät menetelmät (Elberzhager ym., 2017).

Riungu-Kalliosaari ym. (2016) haastattelivat suomalaisia ohjelmistoalanyrityksiä liittyen DevOpsin hyödyntämiseen, tutkimuksen mukaan DevOps mahdollisti uusien ideoiden ja vaatimusten nopean toteuttamisen ja testaamisen, joka nopeutti sidosryhmille luotavan arvon realisoitumista. Lisääntynyttä kommunikointia pidettiin positiivisena asiana, mutta toisaalta todettiin myös, että erityisesti sähköisten viestimien kautta tapahtuva kommunikointi ei korvannut kasvokkain tapahtuvaa kommunikointia, jota pidettiin tärkeänä. Lisäksi kasvaneesta yhteistyöstä huolimatta oli havaittavissa ristiriitoja järjestelmän kehittämiseen ja ylläpitoon liittyvissä tavoitteissa. Vaikkakin kommunikointi nähtiin positiivisena asiana, sitä pidettiin myös yhtenä DevOpsiin liittyvänä haasteena erityisesti organisaatioissa, joissa oli paljon pitkään samoissa työtehtävissä toimineita henkilöitä. Uuden toimintakulttuurin omaksuminen aiheutti muutosvastarintaa. DevOpsin todettiin myös vähentävän kehitystiimiin kohdistuvia paineita, koska automatisoitujen julkaisuprosessien ansioista uusien päivitysten tuotantoon viemisestä tuli rutiininomainen toimenpide (Riungu-Kalliosaari ym., 2016).

3.4 DevOps mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämisessä

DevOpsia pidetään yhtenä keskeisenä tekijänä, joka tukee mikropalveluarkkitehtuuriin perustuvia järjestelmiä helpottamalla itsenäisten mikropalveluiden kehittämistä, hallintaa ja julkaisua. DevOps-käytänteiden hyödyt ilmenevät erityisesti monimutkaisten testaus- ja tuotantoympäristöjen ja julkaisuinfrastruktuurien rakentamisessa. (Balalaie ym., 2016; Zhu ym., 2016 ja Ebert ym., 2016). Callanani ja Splilanen (2016) raportoivat, kuinka organisaatio muunsi vanhan monoliittiseen arkkitehtuuriin perustuvan järjestelmän

mikropalvelupohjaiseksi. Mikropalveluiden määrän kasvaessa pian kuitenkin havaittiin mikropalveluarkkitehtuurin aiheuttavan suuria haasteita järjestelmän julkaisuun liittyen. Vanha useita manuaalisia vaiheita sisältänyt julkaisuprosessi ei soveltunut enää useista mikropalveluista koostuvan järjestelmän julkaisemiseen. Palveluiden manuaalisesta julkaisusta oli tullut liian työläs ja aikaa vievä prosessi, joka oli jopa entiseen arkkitehtuuriin verrattuna hitaampi. Täten yrityksessä päätettiin ottaa käyttöön DevOps-toimintamalli, jonka avulla jokaisen mikropalvelun julkaisu saatiin lopulta automatisoitua siihen pisteeseen, että organisaatio kykeni suorittamaan useita tuotantopäivityksiä päivässä. Automaatio saavutettiin soveltamalla moderneja julkaisuprosessien automatisointiin tarkoitettuja työkaluja ja muodostamalla moniosaajista koostuvia tiimejä, jotka pyrkivät yhteistyön kautta ratkaisemaan järjestelmien julkaisuun liittyvät ongelmat. Järjestelmän jakamisella mikropalveluiksi julkaisuihin käytettävä aika väheni 90%.

Senapathin, Buchanin ja Osman (2018) suorittivat tapaustutkimuksen, jossa kohdeyritys sovelsi DevOps-toimintamallia mikropalveluarkkitehtuuriin perustuvan järjestelmän kehittämisessä. Tutkimuksen mukaan DevOpsin käyttö johti tuottavampaan ja tyytyväisempään tiimiin sekä parempaan asiakaskokemukseen. Pääasiallisina ajureina toimii korostunut yhteistyö, uusien automaatiota tukevien teknologioiden käyttö ja autonominen itseohjautuva tiimi, joka jakoi vastuun järjestelmästä. Uuden toimintamallin omaksuminen aiheutti myös useita haasteita. Vaikkakin uusien teknologioiden käyttöönotto koettiin positiivisena asiana, niiden opettelemiseen kului paljon aikaa, joka oli pois järjestelmän kehitystyöstä. Lisäksi uusien teknologioiden osaajia oli lähes mahdotonta rekrytoida, koska DevOps-toimintamallia edellyttävien teknologioiden osaajia ei ollut vapailla työmarkkinoilla.

DevOpsin yksi oleellisista tavoitteista on jatkuvan julkaisemisen saavuttaminen (Hüttermann, 2012, 112). Sen merkitys korostuu järjestelmän kehitysvaiheessa erityisesti tilanteissa, joissa järjestelmän eri osa-alueita pyritään kehittämään samaan aikaan. Mikropalveluarkkitehtuuriin perustuvassa kehittämisessä erilliset kehitystiimit voivat olla vastuussa järjestelmän eri osista, jotka perustuvat eri teknologioihin, jolloin myös erillisiä palveluita saatetaan julkaista eri menetelmien avulla. Mikropalveluarkkitehtuuriin modulaarisuus ja itsenäisyys toimii myös vastauksena haasteisiin, joita ilmenee ohjelmistokehitysprojekteissa, joissa suuri määrä kehittäjiä työskentelee saman lähdekoodin parissa. Tällainen työskentelytapa aiheuttaa usein konflikteja versionhallintaan lisätyissä muutoksissa, koska kehittäjät saattavat joutua tekemään muutoksia jauttuihin ominaisuuksiin (Adams & McIntosh, 2016). Mikropalveluarkkitehtuuriin perustuvassa järjestelmässä tätä ongelmaa ei ole, koska jokaiselle mikropalvelulle voidaan tarvittaessa rakentaa kokonaan oma julkaisuputki, ja jokainen tiimi on vastuussa vain omiin palveluihinsa tehdyistä muutoksista. Jatkuvan integraation avulla itsenäisesti julkaistavista mikropalveluista saadaan nopea palaute, joka johtaa yhä ketterämpään kehittämiseen. Toinen oleellinen hyöty saadaan tuotantoympäristöihin tapahtuvissa julkaisuissa, jotka voidaan optimaalisessa tilanteessa suorittaa palvelukohtaisesti, jolloin ko-

ko järjestelmän alas ajon edellyttäviltä päivityksiltä voidaan välttyä kokonaan. Mikropalveluarkkitehtuuriin perustuvassa jatkuvassa julkaisemisessa korostuu myös konttiteknologioiden hyödyt, koska yksittäiset mikropalvelut voidaan toteuttaa eri teknologioilla. Dockerin kaltaisten konttijärjestelmien hyödyntäminen mahdollistaa eri teknologiapinoihin perustuvien mikropalveluiden saumattoman julkaisemisen samalle palvelimelle (O'Connor, Elger & Clarke, 2017).

Mikropalveluarkkitehtuuri toimii jatkuvaa integraatiota ja jatkuvaa toimittamista helpottavana arkkitehtuurityylinä. Kehitettävän järjestelmän arkkitehtuuri näyttelee suurta roolia julkaisuinfrastruktuuria rakennettaessa (Elberzhager ym., 2017). Huonosti suunniteltua järjestelmää on vaikea testata, jolloin automaattisesta julkaisusta tulee haastavaa. Chen (2018) suoritti tapaustutkimuksen, jossa selvitettiin DevOpsin käytänteiden soveltuvuutta mikropalveluarkkitehtuuriin perustuvan järjestelmän kehittämisessä. Hänen mukaansa mikropalveluiden suuri määrä asetti teknisiä haasteita jatkuvaa toimittamista tukevaa infrastruktuuria rakennettaessa. Haasteet liittyvät mikropalveluiden testaamiseen ja niiden väliseen kommunikointiin. Oleellisena mikropalveluarkkitehtuurin mahdollistajana pidettiin jatkuvan toimittamisen mahdollistavan infrastruktuurin rakentamista, jonka kautta itsenäisiä mikropalveluita pystyttiin testaamaan ja julkaisemaan. Tästä voidaan myös päätellä, että mikropalveluarkkitehtuuriin perustuvan järjestelmän kehittämien asettaa kehityksen alkuvaiheessa ylimääräisiä haasteita, joista ei välttämättä perinteistä monoliittista järjestelmää kehittäessä tarvitse huolehtia. Näin ollen uutta järjestelmää rakennettaessa mikropalveliden kehittäminen saattaa olla hitaampaa verrattuna muihin arkkitehtuurityyleihin perustuviin järjestelmiin.

Arkkitehtuurin merkitys on havaittu myös monissa muissa ohjelmistojen julkaisemista ja DevOpsia käsittelevissä tutkimuksissa. Forsgren, Humble ja Kim (2018) korostavat, kuinka löyhäkytkentäinen (Loosely Coupled) arkkitehtuuri mahdollistaa tehokkaan jatkuvan toimittamisen. Saman kaltaisiin johtopäätöksiin päätyivät myös Ståhl ym. (2017) tutkiessaan miten organisaatiot soveltavat jatkuvan integraation periaatteita. He erityisesti mainitsivat mikropalveluarkkitehtuurin olevan arkkitehtuurityyli, joka vie modulaarisuuden äärimmilleen ja soveltuu täten erityisen hyvin jatkuvaa toimittamista tavoittelevan organisaation arkkitehtuurityyliksi. Myös Laukkanen ym. (2017) pitivät arkkitehtuurin modulaarisuutta yhtenä ratkaisuna jatkuvaan toimittamisen liittyviin haasteisiin. Taulukosta 2 voidaan nähdä yhteenveto, miten mikropalveluarkkitehtuuri ja DevOps tukevat toisiaan.

TAULUKKO 2 Miten DevOps mikropalveluarkkitehtuuri tukevat toisiaan.

DevOps-periaate	**Mikropalveluarkkitehtuuriin liittyvä haaste**
Yhteistyö	*Mikropalveluiden julkaiseminen*: Korostuneen yhteistyön ansioista DevOps-tiimi pystyy rakentamaan tehokkaita ratkaisuja monimutkaisten järjestelmien julkaisuun ja ylläpitoon. Yhteistyön tarve korostuu erityisesti monimutkaisten tuotantoympäristöjen rakentamisessa.
Automaatio ja työkalut	*Mikropalveluiden julkaiseminen:* Automatisointi mahdollistaa hajautettujen järjestelmien tehokkaan kehittämisen, testaamisen ja julkaisemisen.
Monitorointi ja mittaaminen	*Mikropalveluiden virheiden seuranta ja skaalautuminen:* Hajautettujen järjestelmien tilan seuranta vaatii erityisten työkalujen hyödyntämistä tai kehittämistä.
Jatkuvuus	*Mikropalveluiden rinnakkainen kehittäminen:* Mikropalveluarkkitehtuuriin perustuvan järjestelmän kehittäminen voidaan jakaa usean DevOps-tiimin kesken siten, että jokainen tiimi on vastuussa sille osoitettujen mikropalveluiden kehittämisestä, testaamisesta ja julkaisemisesta. Tämä mahdollistaa tehokkaan jatkuvan integraation ja jatkuvan toimittamisen.

Soldanin ym. (2018) suorittaman mikropalveluarkkitehtuuriin liittyviä haasteita ja hyötyjä kartoittavan tutkimuksen mukaan mikropalveluihin perustuvien järjestelmien operointiin liittyvinä suurimpina haasteina koettiin järjestelmän resurssien käytön sääntely sekä palveluiden monitorointi ja hallinta. Nämä haasteet ovat juuri keskeisiä teemoja, joita DevOpsia hyödyntämällä pyritään ratkaisemaan. Soldanin ym. (2018) löydökset ovat linjassa Ebertin ym. (2017) ja Balalaien ym. (2016) esittämän idean kanssa, jonka mukaan mikropalveluarkkitehtuurin tehokas hyödyntäminen edellyttää DevOps käytänteiden omaksumista. Kuten yllä esitellyistä tapaustutkimuksista voidaan havaita DevOpsin käyttöönotto ei ole yksinkertainen prosessi, vaan se vaatii muutoksia organisaation rakenteessa ja kulttuurissa sekä uusien työkalujen ja prosessien hallintaa. Sen mukana tuomat edut tukevat selkeästi monimutkaisten järjestelmien kehittämistä korostamalla prosessien automatisointia, mutta asettavat samalla useita haasteita uusien toimintatapojen ja kulttuurin omaksumisessa.

4 TUTKIMUKSEN TOTEUTUS

Tietojärjestelmätieteessä tyypillisesti sovelletaan sekä laadullisia että määrällisiä tutkimusmenetelmiä (Myers & Avison, 2002). Tämän työn tutkimusmenetelmänä hyödynnetään laadullista teemahaastattelumenetelmää. Laadullinen tutkimusmenetelmä soveltuu hyvin määrällisesti vaikeasti mitattavissa olevien ilmiöiden tutkimiseen (Hirsjärvi, Remes & Sajavaara, 2009, 161). Näin ollen sen voidaan myös katsoa soveltuvan hyvin mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä tukevien tekijöiden selvittämiseen. Tämän työn tavoitteena on selvittää mikropalveluarkkitehtuurin perustuvien järjestelmien kehittämistä tukevia prosesseja ja toimintamalleja sekä mikropalveluarkkitehtuuriin liittyviä suunnitteluratkaisuja, joiden avulla voidaan realisoida mikropalveluarkkitehtuurin liitetyt arvolupaukset. Kuten kirjallisuuskatsauksessa käy ilmi mikropalveluarkkitehtuuri ei ole tarkkaan määriteltävissä oleva ilmiö, jonka lisäksi sen katsotaan olevan edelleen kehittyvässä vaiheessa oleva arkkitehtuurityyli. Täten laadullisen tutkimuksen avulla voidaan kerätä tietoja ilmiön nykytilasta ja siitä miten kyseistä arkkitehtuurityyliä tällä hetkellä sovelletaan. Tutkielman empiirisessä osuudessa hyödynnetään laadullista teemahaastattelu menetelmää, jonka puitteissa haastatellaan mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämiseen ja suunnittelemiseen erikoistuneita asiantuntijoita. Seuraavissa alaluvuissa esitellään ja perustellaan laadullisen tutkimusmenetelmän valintaa ja esitellään haastatteluissa käytetty teemahaastattelumenetelmä. Lisäksi kuvataan haastatteluiden toteutus sekä aineiston analysoinnissa käytetyt menetelmät.

4.1 Laadullinen tutkimusmenetelmä

Hirsjärven ym. (2009, 164–165) mukaan laadulliselle tutkimukselle on tyypillistä hypoteesittömyys, jonka vuoksi laadullisella tutkimuksella pyritään usein löytämään uutta odottamatonta tietoa tutkittavasta aiheesta. Yksi laadullisen tutkimuksen tavoitteista on löytää käsiteltävästä datasta

yhdenmukaisuuksia ja toistuvia malleja (Hirsjärvi ym., 2009, 164–165). Edellä esitetyn kuvauksen mukaan laadullinen menetelmä soveltuu hyvin vastaamaan tämän tutkielman tutkimuskysymyksiin. Peshkin (1993) jakaa laadullisen tutkimuksen tavoitteet neljään kategoriaan. Laadullisella tutkimuksella voidaan pyrkiä kuvailemaan, tulkitsemaan, vahvistamaan tai arvioimaan tutkimuksen kohteena olevaa ilmiötä. Kuvailevan tutkimuksen tuloksena voi olla esimerkiksi prosessimalli, jonka avulla voidaan kuvata ilmiöitä, josta ei juurikaan ole aiempaa tutkittua tietoa. Tulkitsevassa tutkimuksessa tavoitteena on löytää uusia näkökulmia tutkittavaan ilmiöön, joiden avulla ilmiötä voidaan selittää ja yleistää. Vahvistavaa laadullista tutkimusta voidaan puolestaan soveltaa esimerkiksi hyvin ennalta tiedossa olevien teorioiden testaamiseen erilaisissa ympäristöissä. Arvioivan tutkimuksen avulla taas voidaan tarkastella tiedossa olevaa ilmiötä ja näin ollen pyrkiä tutkimaan, miten se ilmenee eri konteksteissa (Peshkin, 1993). Edellä esitettyjen kuvauksien perusteella tämän työn tavoitteet sijoittuvat kuvailevan ja tulkitsevan tutkimuksen välimaastoon.

Kaplanin ja Maxwellin (2005) mukaan laadullisessa tutkimuksessa dataa voidaan kerätä useiden menetelmien kautta, keräämiseen hyödynnetään tyypillisesti erilaisia haastatteluja, paikan päällä tapahtuvaa tarkkailua sekä aiheeseen liittyvää kirjallista materiaalia.

Laadullisia tutkimusmenetelmiä on sovellettu usein ohjelmistokehittämisprosesseihin ja toimintamalleihin liittyvissä tutkimuksessa (Ståhl & Bosch, 2014; Claps ym., 2015; Lwakatare ym., 2015; Erich ym, 2017). Myös mikropalveluarkkitehtuuria analysoivissa tutkimuksissa on viime aikoina sovellettu laadullisia haastatteluja. Esimerkiksi Taibi ja Lenardruzzi (2018) käyttivät laadullista tutkimusmenetelmää selvittäessään mikropalveluarkkitehtuuriin liittyviä antisuunnittelumalleja. Myös Balalaie ym. (2018) ja Francesco ym. (2018) keräsivät dataa hyödyntäen laadullisia menetelmiä selvittäessään, miten vanhoja järjestelmiä muunnetaan mikropalvelupohjaisiksi. Edellä esitetyt tutkimukset osaltaan vahvistavat laadullisen tutkimusmenetelmän soveltuvuutta myös tämän työn datan keruu menetelmäksi.

Erityisesti tietojärjestelmätieteessä laadullisen tutkimuksen avulla voidaan pyrkiä ymmärtämään uusia nousevia trendejä, joista ei ole olemassa tarkkoja määritelmiä. Laadullisen tutkimuksen avulla pystytään myös syventymään aiheeseen ja ymmärtämään, mitä keskeisiä elementtejä ilmiöön liittyy. Lisäksi voidaan selvittää miten eri konteksteissa toimivat henkilöt kokevat ilmiön ja toisaalta mitä he pitävät merkittävänä ilmiöön liittyen (Kaplan & Maxwell, 2005). Laadullinen haastattelu sallii myös avoimen kanssakäymisen haastateltavan kanssa, mikä mahdollistaa ennalta odottamattoman tiedon löytämisen. Mikäli esimerkiksi haastateltavien näkemykset aiheesta eroavat merkittävästi toisistaan, voidaan haastattelun aikana pyrkiä selvittämään mistä tämä johtuu esittämällä tarkentavia kysymyksiä, jotka eivät kuulu alkuperäiseen haastattelunrukoon.

Laadullisiin haastatteluihin liittyy kuitenkin myös useita haasteita. Myers ja Newman (2007) kuvailevat haastatteluja yhdeksi tärkeimmistä laadullisista tutkimusmenetelmistä. He kuitenkin myös korostavat haastatteluihin liittyviä

haasteita. Haastattelutilanteeseen sisältyy aina useita tekijöitä, mitkä voivat vaikuttaa saatujen vastauksien laatuun. Esimerkiksi kiire, väärinymmärrykset sekä vastaajan tai haastattelijan kokemattomuus keinotekoisista haastattelutilanteista voivat vaikuttaa saatuihin vastauksiin (Myers & Newman, 2007). Tässä työssä haasteiden voidaan katsoa liittyvän haastattelijan kokemattomuuteen laadullisten haastatteluiden toteuttamisesta. Tämän vuoksi haastattelurungon toimivuus varmistetiin suorittamalla testihaastatteluja ennen virallisten haastattelujen aloittamista. Haastattelurungon testaamisella voitiin varmistua haastattelukysymysten toimivuudesta sekä arvioida haastatteluihin kuluvaa aikaa.

4.2 Teemahaastattelu

Tässä tutkielmassa sovelletaan laadullisiin tutkimusmenetelmiin kuuluvaa teemahaastattelumenetelmää. Hirsjärvi ja Hurme (2000, 47–48) kuvailevat teemahaastattelua puolistrukturoiduksi haastattelumenetelmäksi, joka rakentuu tutkittavaan ilmiöön liittyvien teemojen ympärille. Teemahaastattelu ei sisällä ennalta tarkkaan määriteltyjä kysymyksiä, vaan sallii avoimen keskustelun jokaiseen teemaan liittyen. Tämä mahdollistaa haastattelujen lomassa tehtävien tarkentavien kysymysten esittämisen, joiden avulla voidaan löytää uutta arvokasta tietoja, mikä ei tulisi ilmi ennalta tarkkaan rajatussa haastattelussa. Teemahaastatteluissa hyödynnyttävien teemojen valinta perustuu teoreettiseen viitekehykseen, joka luodaan ilmiöistä aiemmin saatavilla olevien tietojen pohjalta. Teemahaastatteluun valitaan henkilöitä, joilla tiedetään jo valmiiksi olevan tutkittaviin teemoihin liittyvää aiempaa tietoa. Näin varmistetaan haastatteluiden avulla saatavan tiedon laatu (Hirsjärvi & Hurme, 2000, 47–48).

Tämän työn empiirisessä teemahaastattelussa selvitettiin mitkä ohjelmistokehittämiseen liittyvät toimintamallit ja prosessit tukevat ja mahdollistavat mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämisen. Lisäksi selvitettiin teknisestä näkökulmasta mitkä tekniset ratkaisut tukevat mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä. Haastatteluiden teemat rakennettiin aiheeseen liittyvien suurten kokonaisuuksien ympärille siten, että ne eivät ohjannet liian yksityiskohtaisesti haastateltavien vastauksia. Teemojen valinta perustui kirjallisuuskatsauksessa tunnistettujen mikropalveluarkkitehtuurin kannalta merkittävinä pidettyihin osa-alueisiin. Haastattelut jakaantuivat kahteen pääteemaan:

- Julkaisuprosessit.
- Kehittämistä tukevat toimintamallit.

Aiemman tutkimuskirjallisuuden pohjalta tunnistettiin kaksi merkittävää tekijää, joiden voidaan katsoa helpottavan mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä. Julkaisuprosesseihin liittyen merkittäviksi tekijöiksi tunnistettiin korkean tason automaation ja jatkuvuuteen perustuvien julkaisumenetelmien rakentaminen, joiden avulla pystytään tehostamaan mikro-

palveluarkkitehtuuriin perustuvien järjestelmien kehittämistä. Merkittävänä toimintamallina pidettiin DevOpsia, jota hyödyntämällä pyritään korostamaan tavallisesti erillään toimivien sidosryhmien välistä yhteistyötä, joka tukee automatisoitujen infrastruktuurien rakentamista. Toisena pääteemana käsiteltiin mikropalveluarkkitehtuurin kannalta tärkeiksi tunnistettuja osa-alueita:

- Mikropalveluiden muodostaminen
- Mikropalveluiden kehittäminen
- Mikropalveluiden välinen kommunikointi
- Datan hallinta
- Virheiden käsitteleminen ja monitorointi
- Skaalautuvuus ja löydettävyys

Kirjallisuuskatsauksessa tunnistettiin useita mikropalveluarkkitehtuuriin liitettyjä suunnittelumalleja ja toimintaperiaatteita. Näiden pohjalta teemoiksi valikoituivat mikropalveluarkkitehtuuriin liittyviä osa-alueita, joiden voidaan katsoa olevan erityisen merkityksellisiä toimivaa mikropalveluarkkitehtuuria rakennettaessa. Mikropalveluiden muodostaminen tunnistettiin yhtenä haastavimpana asiana liittyen mikropalveluarkkitehtuuriin.

Mikropalveluiden kehittämiseen puolestaan liittyy piirteitä, joiden voidaan katsoa tehostavan mikropalveluiden kehittämistä, kuten esimerkiksi mikropalveluiden rinnakkainen ja teknologia riippumaton kehittäminen. Näin ollen on mielenkiintoista selvittää kuinka merkittävinä alan asiantutijat pitävät kirjallisuuskatsauksessa tunnistettuja kehittämiseen liittyviä käytänteitä.

Mikropalveluiden välinen kommunikointi on merkittävässä osassa mikropalveluarkkitehtuuria. Kirjallisuuskatsauksen perusteella siihen liitettiin useita tärkeinä pidettyjä toteutustapoja, joiden avulla pyritään määrittelemään miten mikropalveluiden tulisi kommunikoida keskenään.

Datan hallinnassa puolestaan suurimpien haasteiden todettiin liittyvän mikropalveluarkkitehtuurin hajautettuun luonteeseen, mikä aiheuttaa erityisisä haasteita useiden mikropalveluiden kautta kulkevien operaatioiden käsittelemisessä. Mikropalveluarkkitehtuuri myös mahdollistaa useiden eri tietokantateknologioiden yhdistelemisen sekä hyödyntää suunnittelumallia, minkä mukaan mikropalveluiden tietokannat tulee eriyttää toisistaan.

Kirjallisuuskatsauksen perusteella mikropalveluiden löydettävyyttä pidettiin edellytyksenä mikropalveluiden skaalautuvuudelle. Löydettävyyteen liitettiin kaksi oleellista toimintatapaa, joiden avulla löydettävyys tyypillisesti ratkaistaan. Näin ollen on mielenkiintoista kerätä käytännön kokemuksia siitä, miten palveluiden löydettävyyteen liittyvät ongelmat ovat ratkaistu tuotannossa olevissa järjestelmissä ja mitä haasteita eri ratkaisuihin liittyy.

Kirjallisuuskatsauksen perusteella järjestelmien monitorointia ja virheiden käsittelyä pidettiin mikropalveluarkkitehtuurissa erityisen merkittävänä tekijänä. Mikropalveluarkkitehtuurin hajautetun luonteen vuoksi tilan seuranta ja virheiden käsittely eroaa huomattavasti perinteisistä monoliittisista järjestelmistä tuoden mukanaan uusia haasteita. Täten on mielenkiintoista selvittää mihin

elementteihin monitoroinnissa ja virheiden käsittelyssä kiinnitetään erityistä huomiota ja mitä piirteitä monitorointiin liittyen alan asiantuntijat pitävät merkittävinä. Mikropalveluiden skaalautuvuutta puolestaan pidettään usein mikropalveluarkkitehtuurin tärkeimpänä ominaisuutena ja syynä, miksi juuri suuria käyttäjämääriä tukevat organisaatiot aloittivat kehittämään mikropalveluarkkitehtuuriin perustuvia järjestelmiä. Näin ollen on mielenkiintoista selvittää, mitä pitää ottaa huomioon, kun pyritään rakentamaan skaalautuvia mikropalveluita ja mitä haasteita skaalautuvuuden tavoitteleminen asettaa. Yllä esiteltyjen teemojen lisäksi haastateltavat saivat nostaa esiin mielestään tärkeitä teemoja, jotka ovat ennalta esitettyjen teemojen ulkopuolella.

4.3 Aineiston kerääminen

Haastateltavien löytämisessä hyödynnettiin LinkedIn-palvelua, jonka hakutoiminolla pystyttiin helposti tunnistamaan potentiaalisia haastateltavia. Haastateltavia lähestyttiin lähettämällä viesti LinkedIn-palvelun tarjoaman viestiominaisuuden kautta, jossa esiteltiin tutkija sekä tutkimuksen aihe ja tarkoitus. Yhteyttä otettiin yhteensä 30 henkilöön, joista seitsemän (taulukko 3) päädyttiin haastattelemaan. Haastateltaviksi valikoitui eri tehtävissä toimivia henkilöitä, joilla kaikilla oli vähintään kaksi vuotta kokemusta mikropalveluarkkitehtuuriin perustuvien järjestelmien parissa työskentelystä ja yleisesti kokemusta ohjelmistoalalta toimimisesta vähintään kahdeksan vuotta. Rajaamalla haastateltavien henkilöiden kokemus ohjelmistoalalta ja mikropalveluarkkitehtuurista pyrittiin varmistumaan siitä, että kaikilla haastateltavilla on mikropalveluarkkitehtuurin lisäksi selkeä kokonaiskuva yleisesti ohjelmistokehittämiseen ja eri arkkitehtuurityyleihin liittyvistä haasteista.

TAULUKKO 3 Tutkimuksessa haastatellut henkilöt

#	Työtehtävä	Kokemus ohjelmistoalalta	Kokemus mikropalveluarkkitehtuurista
H1	Pääarkkitehti	20 vuotta	6 vuotta
H2	Ohjelmistoarkkitehti	15 vuotta	2 vuotta
H3	Ohjelmistokehittäjä	8 vuotta	4 vuotta
H4	Ohjelmistokehittäjä	20 vuotta	3 vuotta
H5	Johtava ohjelmistokehittäjä	20+ vuotta	4 vuotta
H6	Ohjelmistoarkkitehti	25 vuotta	3 vuotta
H7	Ohjelmistoarkkitehti	16 vuotta	2 vuotta

Haastatteluiden toteuttamisessa seurattiin Myersin ja Newmanin (2007) ehdottamaa käsikirjoitusta, jonka mukaan haastattelu voidaan jakaa neljään vaiheeseen. Ensimmäisessä vaiheessa haastattelija esittelee itsensä. Toisessa vaiheessa haastattelija esittelee tutkimuksen tarkoituksen ja kertoo, miten ja mihin haas-

tattelujen avulla kerättyä dataa hyödynnetään. Kolmantena vaiheena on itse haastattelun suorittaminen. Viimeisessä vaiheessa haastattelija varmistaa onko haastateltavalla mitään lisättävä ja kysyy, onko haastateltavalla ehdotuksia mahdollisista muista haastateltavista (Myers ja Newman, 2007). Haastattelujen toteuttamisessa käytettiin Skype- ja Google Hangouts-videopuhelusovelluksia ja haastatteluiden nauhoittamisessa käytettiin ApowerRec-sovellusta.

Haastattelut aloitettiin kertomalla haastateltaville, mitä varten haastatteluja ollaan tekemässä sekä miten kerättyä dataa käsitellään. Ennen varsinaisten teemojen käsittelemistä haasteltavia pyydettiin määrittelemään omin sanoin, mikä heidän mielestään on mikropalvelu. Näin varmistuttiin siitä, että haastattelijalla ja haastateltavalla on jaettu ymmärrys käsiteltävästä aiheesta. Haastatteluiden aikana haastateltaville näytettiin käsiteltävät teemat ruudunjako ominaisuutta hyödyntämällä, jolloin sekä haastattelija ja haasteltava pystyivät seuraamaan, mitä teemaa milloinkin käsiteltiin. Teemojen lisäksi haastattelijalla oli käytössä tarkempi runko haastattelujen kulusta, johon oli listattu tunnistettujen teemojen kannalta oleellisiksi katsottuja kysymyksiä (LIITE 1). Haastattelurungosta kuitenkin poikettiin useiden teemojen kohdalla, sillä haasteltavien vastauksiin liittyen haluttiin esittää tarkentavia kysymyksiä, jotka johtivat keskusteluihin, jotka eivät olleet osa haastattelurunkoa. Haastattelujen lopuksi jokaiselta haastateltavilta vielä kysyttiin haluavatko he nostaa esiin ennalta määriteltyyn teemojen ulkopuolisia asioita. Seuraavassa alaluvussa esitellään, miten kerätty aineisto analysointiin.

4.4 Analysointimenetelmä

Hirsjärvi ja Hurme (2000, 138–139) esittävät haastatteluiden avulla kerätyn datan purkamiseen kahta menetelmää. Materiaali voidaan kirjoittaa puhtaaksi kokonaisuudessaan tai sitä voidaan pyrkiä analysoimaan suoraan nauhoitetussa muodossa. Jälkimmäinen vaihtoehto soveltuu erityisesti lyhyiden haastattelujen analysointiin. Mikäli haastattelu halutaan purkaa kokonaisuudessaan, voidaan materiaalin litteroinnin tarkkuus määrittää tutkimuksen tavoitteiden sekä valitun analysointimenetelmään mukaan (Hirsjärvi ja Hurme, 2000, 138–139).

Tässä työssä haastattelujen avulla kerätyn materiaalin läpikäyminen aloitettiin litteroimalla jokainen haastattelu tekstimuotoon. Materiaali litteroitiin teemoittain siten, että jokaiseen teemaan liittyvä keskustelu litterointiin kokonaisuudessaan niin, että puhekielinen keskustelu muutettiin kirjakieleksi. Toimenpiteen katsottiin helpottavan materiaalin jälkeenpäin tapahtuvaa analysointia ja lisäksi tekstin kirjakielelle muuntamisella ei katsottu olevan negatiivisia vaikutuksia datasta esiin nousevien merkityksellisten seikkojen löytämiseen liittyen. Datan litterointi suoritettiin Word-tekstinkäsittelyohjelmalla. Datasta pyrittiin jo litterointivaiheessa nostamaan esiin tärkeiksi tunnustettuja elementtejä korostamalla merkittävinä pidettyjä vastauksia tekstissä. Litteroinnin tuloksena analysoitavaa tekstiä syntyi yhteensä 45 sivua.

Hirsjärven ja Hurmeen (2000, 136) mukaan laadullisen materiaalin analysointiin voidaan soveltaa useita erilaisia tapoja. Tässä työssä kerätyn datan analysoimiseen sovellettiin teorialähtöistä sisällönanalyysiä. Tuomen ja Sarajärven (2010, 112) mukaan teorialähtöisessä sisällönanalyysissä hyödynnetään aiemmin aihetta käsitelleiden auktoriteettien ajatuksia, joiden pohjalta kerätty data analysoidaan. Tämän työn teoreettisena viitekehyksenä toimi kirjallisuuskatsauksen avulla tunnistetut teemat, joiden katsottiin olevan merkityksellisiä mikropalveluarkkitehtuuriin pohjautuvia järjestelmiä rakennettaessa.

Kaplan ja Maxwell (2005) esittelevät laadullisen tutkimusmateriaalin analysoimiseen koodausmenetelmän, jota myös sovellettiin tämän työn analysoinnissa. Koodauksen tarkoituksena on tunnistaa kerätystä datasta luokkia, joihin voidaan sijoittaa datasta löydettyjä yhtäläisyyksiä, jotka vastaavat asetettuihin tutkimuskysymyksiin (Kaplan ja Maxwell, 2005). Koodauksen toteuttamiseen Auerbach ja Silverstein (2003) ehdottavat kerätyn datan ryhmittelyä ja jokaisen tunnistetun ryhmän käsittelyä omana kokonaisuutenaan. Tämä helpottaa yhdenmukaisuuksien löytämistä sekä nopeuttaa materiaalin läpikäymistä (Auerbach & Silverstein, 2003). Yllä esiteltyä ajatusta seuraten tässä työssä haastattelujen jäsentämisessä hyödynnettiin haastattelurungon muodostamisessa käytettyjä teemoja, joiden avulla haastateltavien vastaukset saatiin helposti kategorisoitua. Datan analysointia varten litteroidut haastattelut kerättiin yhteen tekstidokumenttiin siten, että kaikkien haastateltavien vastaukset ryhmiteltiin teemoittain. Datan analysointi suoritettiin lukemalla jokaiseen teemaan liittyvät vastaukset läpi, samaan aikaan etsien yhdenmukaisuuksia ja usein esille nousevia oleellisia aiheita ja termejä, joiden kautta pyrittiin luomaan jokaiseen teemaan liittyviä luokkia. Taulukossa 4 esitellään analyysiin kautta esiin nousseet luokat teemoittain.

TAULUKKO 4 Analyysin tuloksena tunnistetut luokat

Teemat	Luokat
Julkaisuprosessit	Jatkuva toimittaminen ja jatkuva integraatio, Kontit
Kehittämistä tukevat toimintamallit	Yhteistyö, Kommunikaatio, Monitorointi, Läpinäkyvyys
Mikropalveluiden muodostaminen	DDD-suunnittelu, Skaalautuminen, Itsenäisyys, Muutos, Haastavuus
Mikropalveluiden kehittäminen	Monikielisyys, Haastavuus, Ketteryys
Mikropalveluiden välinen kommunikointi	REST-periaatteet, Tapahtumapohjainen arkkitehtuuri, Standardointi, Yhtenäinen rajapinta
Datan hallinta	Tietovarastojen eriyttäminen, Teknologia riippumattomuus
Virheiden käsittely ja monitorointi	Alustojen hyödyntäminen, Keskitetty monitorointi
Skaalautuminen ja löydettävyys	Alustojen hyödyntäminen, Kontit

Tunnistettujen luokkien perusteella voidaan jo havaita, että mikropalveluiden kehittämiseen tällä hetkellä hyödynnettävät käytänteet ovat hyvin linjassa kirjallisuuskatsauksen kautta tunnistettujen elementtien kanssa. Seuraavassa luvussa esitellään tutkimuksen tulokset.

5 TUTKIMUKSEN TULOKSET

Tutkimuksen tulokset esitellään samassa järjestyksessä, missä ne käsiteltiin haastatteluissa (LIITE 1). Ensin käydään läpi mikropalveluarkkitehtuurin yhteydessä hyödynnettäviin julkaisuprosesseihin ja toimintamalleihin liittyvät tulokset, minkä jälkeen käsitellään mikropalveluarkkitehtuuriin liittyvät tulokset. Haastateltaviin viitataan termeillä H1, H2 jne., edellisessä luvussa esitellyn taulukon 4 mukaan.

5.1 Mikropalveluiden määritelmä

Jokainen haastattelu aloitettiin kysymällä haastateltavilta, miten he määrittelevät, mikä on mikropalvelua. Haastateltavien määritelmien mukaan mikropalvelut ovat pienikokoisia, itsenäisiä, pilviympäristöissä ajettavia ohjelmistoja, joiden kehittäminen voidaan hajauttaa useiden tiimien kesken.

5.2 Julkaisuprosessit

Kaikki haastatteluun osallistuneet painottivat jatkuvan integraation ja jatkuvan toimittamisen mahdollistavan infrastruktuurin merkitystä mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämisessä. Automaation merkityksestä mikropalveluiden kehittämisessä keskusteltaessa esimerkiksi H1 ja H4 käyttivät termiä ”välttämätön” ja H2 termiä ”edellytys”, mikä omalta osaltaan vahvistaa ja tukee jatkuvuuteen perustuvien julkaisuprosessien merkityksellisyyttä. H6 mukaan ainoa manuaalinen asia julkaisuprosessissa tulisi olla tuotantoon julkaiseminen. Kaikki muut julkaisemiseen liittyvät operaatiot ovat suositeltavaa automatisoida, niin pitkälle kuin mahdollista. Näin ollen voidaan todeta, että alan asiantuntijoiden vastaukset liittyen kehittämistä tukeviin julkaisuprosesseihin olivat erittäin hyvin linjassa aiemman aihetta käsittelevän kirjallisuuden kanssa. Haastateltavat H1 ja H2 perustelivat mikropalveluiden julkaisupro-

sessien korkean tason automatisointia virheiden välttämisellä sekä työmäärän pienentämisellä. H1:n mukaan pienessäkin mikropalveluarkkitehtuuriin perustuvassa järjestelmässä voi olla helposti 10–20 mikropalvelua. Mikäli julkaisuautomaatio ei ole palveluiden määrän edellyttämällä tasolla, pääsevät virheet helposti kertaantumaan.

Haastatteluissa nousi myös esiin mikropalveluarkkitehtuurin hyödyt julkaisuprosesseihin liittyen. H3 ja H4 nostivat erityisesti esiin mikropalveluiden hyödyt verrattuna monoliittisiin järjestelmiin ja pitivät erittäin merkittävänä mahdollisuutta päivittää vain pieniä osia järjestelmästä. Näin pystytään välttymään pitkäkestoisilta päivitysprosesseilta ja toisaalta myös kyetään todentamaan nopeasti mistä epäonnistunut julkaisu johtuu. H1 nosti esiin myös erityisenä haasteena jatkuvaan julkaisemiseen liittyen vanhat käytänteet ja asiakkaiden kanssa tehdyt sopimukset, joiden vuoksi sopimustekniset syyt rajoittavat julkaisusyklien tiheyttä vaikkakin teknologiset kyvykkyydet olisivat kunnossa.

Konttien käyttö korostui julkaisuprosesseissa. Esimerkiksi H3 kertoi käyttäneensä kaikissa kehittämissään mikropalveluarkkitehtuuriin perustuneissa projekteissa konttipohjaisia julkaisuprosesseja. H1, H2, H3 ja H5 pitivät konttien avulla saavutettavina hyötyinä mikropalveluiden eristämistä kehitysalustasta sekä konttipohjaisten järjestelmien helppoa siirreltävyyttä ja nopeaa pystyttämistä. Toisaalta H4 myös mainitsi, että kaiken, jonka voi tehdä konteilla voi tehdä myös erilaisilla julkaisuskripteillä, mutta totesi myös konttien käytön suoraviivaistavan julkaisuprosessia ja piti suuntausta hyvänä. Myös H7 vertaili kokemuksiaan erilaisten julkaisuskriptien ja konttien käytöstä ja totesi konttien mahdollistavan huomattavasti helpommin hallittavissa olevan julkaisuprosessin.

5.3 Kehittämistä tukevat toimintamallit

Kehittämisessä hyödynnettäviin toimintamalleihin liittyen ennalta oli odotettavissa, että DevOpsia sovellettaisiin laajasti. Haastateltavat nostivat esiin ja pitivät erittäin merkittävinä usein DevOpsin nimissä suoritettavia toimenpiteitä ja periaatteita, vaikkakin ainoastaan H3 ja H5 kertoivat toimivansa varsinaisessa DevOps-tiimissä. Tärkeinä erityisesti DevOpsiin liitettävinä elementteinä haastatteluissa nousi esiin, yhteistyö, automaatio, kommunikointi ja läpinäkyvyys. H3 kuvaili yhteistyötä operoinnista vastaavien henkilöiden kanssa tärkeäksi erityisesti silloin, kun kyseessä oli suurenluokan projekti, joka saattoi sisältää satoja mikropalveluita. Hän myös korosti, kuinka yhteistyön on oltava mukana projektien alusta alkaen, sillä yhtenäisten julkaisuprosessien rakentamien korostuu sitä enemmän, mitä suuremmasta järjestelmästä on kyse. Myös H2:n mukaan järjestelmien operoinnista ja ylläpidosta vastuussa olevien henkilöiden tulisi olla mukana mikropalveluita rakennettaessa projektin alusta asti, sillä mikropalveluiden kehittäminen tuo mukaan useita haasteita, jotka saattavat olla myös operoinnista ja ylläpidosta vastuussa oleville henkilöille täysin odottamattomia. Tällöin on tärkeää, että

kaikki oleelliset sidosryhmät ovat sanomassa mielipiteensä arkkitehtuuriin ja teknologioihin liittyvissä valinnoissa projektin alusta alkaen. H1 puolestaan piti termiä DevOps epäselvänä, mutta piti kuitenkin DevOpsin merkittävänä ominaisuutena siiloutumisen välttämistä, jolloin kommunikointi sidosryhmien välillä saadaan pidettyä nopeana.

Mikropalveluiden kehittämiseen liittyen haastattelevat nostivat esiin infrastruktuuria tukevien työkalujen käytön. H1 korosti valmiiden työkalujen käyttöä. Hänen mukaansa nykyään ei ole enää järkevää, saati mahdollista pyrkiä kehittämään omia ratkaisuja jokaiseen ongelmaan. H4 ja H5 nostivat DevOpsiin liittyen tärkeänä seikkana esiin nopean palautteen saamisen tuotantoympäristöistä ja pitivät erittäin merkittävänä läpinäkyvyyttä liittyen tuotantoympäristöjen monitorointiin. H4 ja H5 korostivat erityisesti, kuinka läpinäkyvän monitoroinnin avulla henkilöt, joilla on eniten tietoa järjestelmän toiminnasta pääsevät nopeasti käsiksi monitoroinnin avulla tuotettuun dataan. Myös H6 toi esiin jaetun vastuun merkityksen ja piti erittäin tärkeänä sitä, että järjestelmän kehittäjillä ja operoijilla on jaettu vastuu tuotantoympäristöjen toimivuudesta. H5 korosti automaation tärkeyttä projektien ylläpitovaiheessa, jolloin julkaisuja ei enää tapahdu tiheään tahtiin. Näin myös vanhempien järjestelmien julkaisu saadaan pidettyä yksinkertaisena, sillä monien manuaalisten vaiheiden hallinta johtaa harvakseltaan julkaistavassa järjestelmässä virheisiin.

Haastateltavien vastauksista voidaan havaita, että mikropalveluita kehitettäessä kommunikointi sidosryhmien välillä korostuu, kuten myös jaettu vastuu, joka edellyttää projektiin osallistuvien sidosryhmien läheistä yhteistyötä.

5.4 Mikropalveluiden muodostaminen

Mikropalveluiden muodostamiseen liittyen esiin nousi odotetusti DDD-suunnittelu periaatteiden hyödyntäminen sekä toisistaan riippuvien ja riippumattomien kokonaisuuksien tunnistaminen. Haastateltavat H1 ja H3 nostivat esiin DDD-periaatteet ja ehdottivat kehitettävän järjestelmän aihealueeseen liittyvien, toisistaan riippuvien kokonaisuuksien tunnistamista (Bounded Context), joiden avulla voidaan päätellä mitkä toiminnot sijoitettaan mihinkin mikropalveluun. H2 ja H4 korostivat mikropalveluiden muodostamiseen liittyen, kuinka tärkeää on pyrkiä tunnistamaan mitkä osa-alueet järjestelmästä tulevat vaatimaan erillistä skaalautuvuutta. Näin ollen näistä toiminnoista tulisi vähintäänkin muodostaa omat mikropalvelut. Tai toisaalta mitkä osat tulevat todennäköisesti vaatimaan eniten muutoksia tulevaisuudessa. Vaikkakaan H4 ja H2 eivät ottaneet suoraan puheeksi DDD-periaatteita, he toivat esiin kehitettävän järjestelmän toimintakontekstien ymmärtämisen, joka nähtiin erityisen tärkeänä mikropalveluita suunniteltaessa.

H1 korosti hyvän palvelujaon löytämisen haastavuutta ja piti sitä jopa haastavimpana asiana koko mikropalveluarkkitehtuurissa. Tämä tunnistettiin myös kirjallisuuskatsauksen perusteella yhdeksi mikropalveluarkkitehtuurin suurimmista haasteista.

Haastateltavat H1, H2, H3 ja H4 nostivat myös esiin ajatuksen, jonka mukaan kokonaan uuden järjestelmän rakentaminen saattaa olla järkevämpää aloittaa kehittämällä monoliittinen järjestelmä, joka voidaan jakaa mikropalveluihin sen jälkeen, kun järjestelmän sisäiset riippuvuudet pystytään ymmärtämään yksiselitteisemmin. Monoliitin rakentamiseen liittyen H1 nosti myös esiin haasteen siitä, että sen rakentamista voi olla vaikea perustella sidosryhmille, jos tavoitteena on lopulta jakaa järjestelmä mikropalveluiksi. Hänen mukaansa mikropalveluarkkitehtuurissa monoliitin rakentaminen on kuitenkin tunnistettu potentiaaliseksi vaihtoehdoksi erityisesti tilanteissa, joissa kehitystiimillä ei ole kokemusta aihealueesta johon järjestelmää ollaan rakentamassa.

H4 korosti myös, kuinka tärkeää skaalautumistarpeen arviointi on mikropalveluarkkitehtuuriin suunnitteluvaiheessa. Näin voidaan varmistaa se, että erillistä skaalautuvuutta vaativat osa-alueet sijoitetaan omiksi mikropalveluikseen. Toisena tärkeänä seikkana H4 nosti esiin mikropalveluiden muutostarpeen tunnistamisen, näin ollen usein muuttavat ominaisuudet tulisi sijoittaa omiksi mikropalveluikseen.

Mikropalveluarkkitehtuurin suunnittelemisen haastavuudesta kertoo myös se, että haastateltavilla oli vaikeuksia määritellä, millä perusteilla voidaan rajata mikropalveluiden koko siten, että ne eivät pääse kasvamaan liian suuriksi. H6 ja H2 antoivat laadullisen kriteerin yhden mikropalvelun laajuudesta. Heidän mukaansa mikropalvelua kehittävän henkilön on pystyttävä ymmärtämään ja selittämään palvelun toiminta. H3 nosti myös esiin tärkeän seikan liittyen palveluiden kokoon. Pieni kokoisten mikropalveluiden avulla vältytään suurilta koodi refaktoroinneilta, jotka usein aiheuttavat haasteita suureen jaettuun koodipohjaan perustuvissa järjestelmissä.

5.5 Mikropalveluiden kehittäminen

Mikropalveluarkkitehtuuriin pohjautuvien järjestelmien kehittämisestä keskusteltaessa esiin nousivat mikropalveluiden monikielisyys, useiden palveluiden rinnakkainen kehittäminen sekä mikropalveluarkkitehtuurin haastavuus. Mikropalveluarkkitehtuurin mahdollistamaa monikielistä kehittämistä pidettiin merkittävänä etuna, mutta siihen liitettiin myös useita haasteita. H1 piti mikropalveluiden monikielisyyttä hyvänä asiana, sillä sen avulla on mahdollista saada toteutettua haluttu toiminnollisuus tilanteeseen parhaiten soveltuvalla ohjelmointikielellä. Hän kuitenkin myös korosti, että organisaatiolla on oltava vahva osaaminen valituista, ohjelmointikielistä sekä riittävä määrä henkilöitä, jotka pystyvät tarvittaessa ottamaan vastuun järjestelmän eri osa-alueista. Myös H2:n mielestä mikropalveluarkkitehtuurin rakentamisessa hyödynnettävien ohjelmointikielien valitsemisessa pitäisi kiinnittää huomioita siihen, että valittu ohjelmointikieli on aktiivisesti kehittyvä kieli. Näin ollen sillä ei ole juurikaan merkitystä, mikä kieli valitaan. H6 piti myös monikielisyyttä hyvänä asiana sen tuoman vapauden vuoksi. Hän ei kuitenkaan pitänyt valittua ohjelmointikieltä kovinkaan merkittävänä asiana, sillä verkkoyhteyden kautta tapahtuva kom-

munikointi piilottaa mikropalveluiden toteutuksen. H7 nosti esiin, kuinka monikielisyys korostuu etenkin silloin, kun tehdään suurenluokan järjestelmiä, joissa on useita palveluita, mitkä vastaavat suuresta määrästä vaihtelevaa toiminnallisuutta. Näin ollen järjestelmän toimintaa voidaan optimoida parhaiten tilanteeseen soveltuvalla ohjelmointikielellä.

Mikropalveluarkkitehtuurin haastavuus nousi esiin mikropalveluiden kehittämisestä keskusteltaessa. H1, H4, H5 ja H6 pitivät erityisen tärkeänä, että järjestelmää kehitettäessä mukana olisi, joku henkilö, joka vastaa kokonaisarkkitehtuurista, ja jolla olisi myös aiempaa kokemusta mikropalveluiden kehittämisestä. Näin erillisiä mikropalveluita kehitettäessä voidaan taata, että kokonaiskuva järjestelmän toiminnasta säilyy. H2:n ja H4:n mukaan mikropalveluiden kehittäminen on alussa hidasta, etenkin mikäli kyseessä on kehitystiimi, joka on tekemässä ensimmäistä mikropalveluarkkitehtuuriin perustuvaan järjestelmäänsä. Rajapintojen ja mikropalveluiden kehittämiseen hyödynnettävien projektipohjien luomisen jälkeen kehittäminen muuttuu nopeaksi ja useita palveluja voidaan kehittää rinnakkain. H3, H6 ja H7 nostivat esiin merkittävänä etuna palveluiden itsenäisyyden ja pienen koon, jonka ansioista yksikin kehittäjä pystyy helposti ymmärtämään yhden mikropalvelun toiminnallisuuden ja näin ollen kehittämään siihen liittyviä ominaisuuksia hyvin nopeasti.

H4:n ja H7:n mielestä kehittämiseen liittyviä haasteita ilmeni erityisesti niiden kehittäjien keskuudessa, jotka olivat tottuneet kehittämään synkronisesti toimivia työpöytäsovelluksia, jolloin mikropalvelufilosofian sisäistäminen aiheutti haasteita. Heidän mukaansa mikropalveluiden itsenäisyyden säilyttämiseen pitäisikin kiinnittää erityistä huomiota koko kehitysprosessin ajan. Mikäli näin ei tehdä pääsee syntymään tilanteita, joissa palveluiden julkaiseminen ja riippumaton kehittäminen heikkenevät ja mikropalveluarkkitehtuuriin oleelliset hyödyt menetetään. Pahimmassa tapauksessa saatetaan päätyä tilanteeseen, jossa mikropalveluarkkitehtuuriin perustuva järjestelmä muistuttaa toisistaan riippuviin palveluihin perustuvaa monoliittista järjestelmää.

5.6 Mikropalveluiden välinen kommunikointi

Mikropalveluiden väliseen kommunikointiin liittyen HTTP-yhteyden kautta tapahtuvaa REST-periaatteita seuraavaa kommunikointia pidettiin tärkeänä. Erityisesti mikropalveluiden tilattomuuden merkitystä korostettiin. Myös kaikki haastatteluihin osallistuneet, pitivät mikropalveluarkkitehtuuriin perustuvan järjestelmän tarjoamista yhtenäisen rajapinnan (API Gateway) kautta merkittävänä ja hyvänä suunnittelumallina. H7 perusteli yhtenäisen rajapinnan yksinkertaistavan järjestelmän käyttöliittymän kehittämistä, koska käyttöliittymän tarvitsee olla tietoinen ainoastaan yhdestä rajapinnasta. H3 toi esiin myös yhtenäisen rajapinnan sijasta käytettävän vaihtoehdon, jossa jokaista asiakassovellusta varten pystytetään oma mikropalvelu, joka kokoaa kunkin asiakassovelluksen vaatiman datan. Tällaista mallia sovellettiin erityisesti tilanteissa, joissa mikropalveluarkkitehtuuriin perustuvaan järjestelmään haluttiin olla yhteydessä eri-

laisilla päätelaitteilla. Näin ollen esimerkiksi web-käyttöliittymän ja mobiilisovelluksen kautta saapuvat kyselyt ohjattiin järjestelmään omien rajapintojensa kautta. Mallin avulla eri asiakassovellusten tarpeet voitiin huomioida ja niille voitiin tarjoilla erikseen kontekstin vaatimaa dataa.

H1, H4 ja H6 ehdottivat tapahtumapohjaisen arkkitehtuurin soveltuvan erityisen hyvin mikropalveluiden väliseen kommunikointiin. H6 kuvaili tapahtumiin perustuvaa kommunikointia kommunikointityyliksi, jossa jokainen järjestelmän läpi virtaava tapahtuma tallennetaan. Esimerkiksi verkkokaupassa tapahtuvaan tilaukseen saattaa liittyä useita tapahtumia, joiden tilaa voidaan muuttaa ainoastaan luomalla uusia tapahtumia. Näin ollen kaikki tilaukseen liittyvät tapahtumat läpikäymällä voidaan selvittää tilauksen viimeisin tila. H4 kertoi myös hybridimallista, jossa osa järjestelmään kuuluvista mikropalveluista kommunikoi viestijonojen kautta ja osa hyödynsi REST-rajapintoja. Hänen mukaansa viestijonojen kautta tapahtuvaa kommunikointia tulisi soveltaa erityisesti silloin, kun järjestelmän sisäiset mikropalvelut joutuvat kommunikoimaan keskenään. H6 korosti kommunikoinnissa myös kyselyiden koostamisen suunnittelemisen tärkeyttä. Mikropalveluarkkitehtuurissa päädytään tilanteisiin, joissa joudutaan suorittamaan useita verkon yli tapahtuvia kyselyjä. Näin ollen olisi tärkeää, että viestit saataisiin koostettua siten, että ne olisivat yksinkertaisia, mutta kuitenkin pystyvät välittämään vaaditun datan mahdollisimman vähäisillä kyselyillä.

H6 nosti esille rajapintojen standardisoinnin tärkeyden, sillä useiden mikropalveluiden samanaikainen kehittäminen vaatii ennalta tarkkaan suunniteltuja kommunikointitapoja. Standardoinnilla hän nimenomaan tarkoitti sitä, että palveluiden kesken välitettävät viestit jakaisivat standardoidun muodon, jossa määritellään esimerkiksi, mitä metadataa välitetään varsinaisen kuorman lisäksi. Lisäksi H5 korosti, että kehityksen edetessä muutokset on pystyttävä suorittamaan olemassa oleviin rajapintoihin siten, että kaikki muutokset ovat taaksepäin yhteensopivia. Näin voidaan välttyä ketjuuntuvilta muutosvaatimuksilta, kun yhden mikropalvelun rajapintaa päivitetään.

5.7 Datan hallinta

Data käsittelemiseen liittyvä keskustelu avattiin käsittelemällä kirjallisuuskatsauksessa tunnistettua mikropalveluarkkitehtuurissa yleisesti hyödynnettyä suunnittelumallia, jossa jokaiselle mikropalvelulle olisi oma tietokanta. H1:n mielestä tämän mallin seuraaminen näin jyrkästi ei ole aina tarpeellista. Hän erityisesti korosti, että jokaiseen tietokantaan pitäisi olla ainoastaan yksi dataa kirjoittava mikropalvelu, mutta sama dataa voisi noutaa useampi mikropalvelu. Näin saadaan vähennyttyä verkon yli tapahtuvaa kommunikointia. H1 nosti esiin myös tietovarastojen jakamiseen liittyvän haasteen. Tietovarastojen jakaminen usean mikropalvelun kanssa johtaa tilanteeseen, jossa tietokantamallin muuttaminen aiheuttaa muutoksia myös kaikissa, sitä lukevissa mikropalveluissa. Tämän vuoksi tulisikin kiinnittää erityistä huomiota, milloin yllä esitel-

tyä tapaa sovelletaan. H3 ja H4 puolestaan korostivat, että mikropalvelut voivat käyttää samaa tietokantaa, mutta niiden pääsy tietokantatauluhin tulee rajoittaa mikropalveluiden liiketoimintavaatimusten mukaan. Näin mikropalveluiden itsenäisyys saadaan säilytettyä, vaikka tietokantamallia muutettaisiin. H4 kuitenkin myös korosti, että saman tietokannan eri tauluihin hajautettu data voi johtaa helposti tietokannan väärinkäyttöön.

Mikropalveluiden datavarastojen omistaminen nousi merkittäväksi aiheeksi datana hallinnasta keskusteltaessa. H1, H2 ja H7 mukaan tietokantojen mikropalvelukohtainen eriyttäminen on tavoiteltava tila mikropalveluarkkitehtuuria rakennettaessa. H2 ja H7 myös mainitsivat, että vahoja järjestelmiä modernisoitaessa mikropalvelupohjaisiksi tietokannat saattavat sisältää sellaisia riippuvuuksia, että olemassa olevan datan jakaminen on liian työlästä, jonka vuoksi mikropalvelut joutuvat käyttämään samaa tietokantaa. H4 nosti esiin tietokantojen eriyttämisen merkittävänä skaalautumista ja suorituskykyä parantavana tekijänä, sillä datan määrän kasvaessa tietokantaan tehtyjen kyselyjen kesto kasvaa. Näin ollen datan hajauttaminen useisiin itsenäisiin tietokantoihin mahdollistaa nopeammat kyselyt. Tietokantojen hajauttamisella voidaan saavuttaa erityistä etuja järjestelmissä, jotka tukevat suuria käyttäjämääriä. H7 piti myös merkittävä etuna useiden eri tyyppisten tietokantaratkaisujen yhdistelmistä mikropalveluarkkitehtuurissa. Esimerkiksi yhdistämällä NoSQL-tietokanta useita kirjoitusoperaatiota suorittavaan mikropalveluun pystytään lisäämään järjestelmän suorituskykyä.

Hajautetut transaktiot tunnistettiin kirjallisuuskatsauksessa yhdeksi keskeiseksi mikropalveluarkkitehtuurin haasteeksi. Myös H1, H3 ja H4 pitivät hajautettujen transaktioiden käsittelemistä suureena haasteena, mikä liittyy mikropalveluarkkitehtuuriin. H3:n ja H4:n mukaan hajautettujen transaktioiden syntymistä tulisi välttää suunnittelemalla mikropalveluarkkitehtuuri siten, että kaikki yhteen operaation sisältyvät toiminnot saadaan sijoitettua aina yhden mikropalvelun sisään. H1:n mukaan hajautettujen transaktioiden käsitteleminen on huomioitava, ennen kuin varsinainen kehittäminen alkaa ja niiden käsittelemiseen on luotava strategia, joka vastaa järjestelmän datan eheydelle asetettuja vaatimuksia. H1 ja H2 esittivät hajautettujen transaktioiden käsittelemiseen orkestrointipalveluiden luomista, joiden avulla seurataan transaktioon osallistuvien mikropalveluiden suoriutumista, jolloin virhetilanteiden sattuessa luodut muutokset pystytään peruuttamaan.

5.8 Virheiden käsittely ja monitorointi

Virheiden käsittelyyn liittyen H1 ja H3 nostivat esiin katkaisija-suunnittelumallin käytön. H1 myös korosti, että virheiden käsittelyyn on olemassa nykyään valmiita palveluverkko ratkaisuja (Service mesh), joiden avulla voidaan suojautua mikropalveluiden välillä tapahtuvilta virhetilanteilta. Palveluverkkojen avulla jokaisen mikropalvelun yhteyteen julkaistaan välityspalvelimena toimiva palvelu, joka vastaa virheiden käsittelemisestä ja muusta liike-

toimintalogiikasta riippumattomasta toiminnallisuudesta. Näin ollen kaikki mikropalveluun saapuvat viestit kulkevat niiden yhteyteen julkaistujen välityspalvelimien kautta, jotka sisältävät esimerkiksi katkaisija-suunnittelumallin mukaisia toimintoja. H1 mainitsi erityisesti Istio-palveluverkko ratkaisun, jota voidaan hyödyntää Kubernetesin yhteydessä. Yleisesti ketjuuntuvien virhetilanteiden välttämiseksi ehdotettiin järjestelmän suunnittelemisesta siten, että niitä ei pääsisi syntymään. Tähän pystytään vaikuttamaan mikropalveluarkkitehtuurin palveluiden jakamista ja palveluiden välisiä kommunikointi strategioita suunniteltaessa.

H1:n, H2:n, H3:n ja H7:n mukaan monitorointi tulisi nykyään toteuttaa pilvialustojen tarjoaminen toimintojen avulla, jolloin vältytään manuaaliselta työltä sekä saavutetaan korkean tason automaatio. Pilvialustojen tarjoamien monitorointi ratkaisuiden avulla voidaan tarkkailla kokonaisvaltaisesti mikropalveluista koostuvan järjestelmän tilaa. Mikropalveluiden monitorointiin liittyen H1:n, H3:n ja H4:n mukaan mikropalveluiden pitää pystyä tarjoamaan rajapinta, minkä kautta sille voidaan lähettää yksinkertainen viesti, johon mikropalvelu vastaa aina samalla tapaa. Tällaisen rajapinnan kautta voidaan varmistua siitä, että mikropalvelu on toimintakuntoinen ja pystyy käsittelemään sille saapuvia pyyntöjä. H3:n mukaan mikropalveluiden tilan seurannan tulee olla automaatista ja poikkeuksista on saatava myös automaattinen ilmoitus esimerkiksi sähköpostin muodossa. H1 esitti myös ajatuksen, jonka mukaan jokaisen mikropalveluarkkitehtuuriin perustuvan järjestelmän läpi kulkeva komento on pysyttävä jäljittämään täydellisesti. Tämä helpottaa vian etsintää ja liittyy myös oleellisesti mikropalveluiden välisen kommunikoinnin standardointiin. Jokaiseen viestiin voidaan esimerkiksi sisällyttää tunniste, joka säilyy muuttumattomana kulkiessaan usean mikropalvelun lävitse. Monitorointiin liittyen erityisen tärkeänä pidettiin myös lokitietojen keskitettyä tallentamista, jolloin niiden seuranta on huomattavasti helpompaa.

5.9 Skaalautuvuus ja löydettävyys

Kaikilla haastateltavilla ei ollut käytännön kokemuksia järjestelmän skaalautuvuuteen ja löydettävyyteen liittyen. Esimerkiksi H6 ja H7 kertoivat, että erityisiä skaalatumistarpeita ei ole ollut ja mikropalveluarkkitehtuuri on valikoitunut käytettäväksi arkkitehtuurityyliksi sen muiden ominaisuuksien vuoksi. Haastatteluissa tuli ilmi, että ennen kuin tarvitaan skaalautuvuutta, mikropalveluihin kohdistuvan kuorman tulee olla melko suuri. Ne haastateltavat, joilla oli kokemuksia skaalautumisesta, olivat hyvin yksimielisiä siitä, että skaalautuvuuteen liittyviä ominaisuuksia ei tule pyrkiä itse kirjoittamaan ohjelmakoodiin, vaan ne tulee toteuttaa hyödyntämällä valmiita ratkaisuja. Esimerkiksi H1 piti skaalautuvuutta alkuperäisenä syynä mikropalveluarkkitehtuurin soveltamiselle. Haastatteluissa tärkeimmiksi skaalautuvuuden mahdollistaviksi tekijöiksi nousivat teknisestä näkökulmasta mikropalveluiden tilattomuuden säilyttäminen sekä operointiin liittyen pilvipohjaisten julkaisualustojen ja työkalujen

hyödyntäminen. Niiden avulla järjestelmän monitorointi ja löydettävyys saadaan automatisoitua, mikä taas mahdollistaa automaattisten skaalautumisen.

Haastateltavat esittivät löydettävyyden ratkaisemiseksesi valmiiden pilvialustojen ja työkalujen soveltamista. Esimerkiksi H2, H3, H4 nostivat esiin pilvialustojen hyödyntämisen, jotka tukevat Dockerin ja Kubernetesin kaltaisia työkaluja. H1:n mukaan mikropalveluiden löydettävyyteen liittyvät ongelmat voidaan ratkaista, joko hyödyntämällä kontteja ja niiden hallintaan tarkoitettuja alustoja tai enemmän manuaalista työtä vaativilla palvelurekisteri ratkaisuilla, kuten Netflixin Eureka. Jälkimmäisessä ratkaisussa jokaiseen mikropalveluun tarvitsee kirjoittaa logiikka, jonka kautta se kommunikoi palvelurekisterin kanssa. Haastatteluiden perusteella mikropalveluiden löydettävyyttä ei juurikaan pidetä enää haasteena, vaan sen ratkaisemiseksi voidaan hyödyntää tunnettuja avoimeen lähdekoodiin perustuvia ratkaisuja.

6 POHDINTA

Tässä luvussa esitellään kuinka saadut tulokset suhteutuvat aiempaan tutkimukseen. Tutkimuksen tavoitteena oli selvittää mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä tukevia tekijöitä. Tutkimuksessa pyrittiin vastaamaan seuraaviin tutkimuskysymyksiin:

- Millä toimintamalleilla ja prosesseilla voidaan tehostaa mikropalveluarkkitehtuuriin perustuvaa kehittämistä?

- Minkälaisiin ratkaisuihin mikropalveluarkkitehtuuria hyödyntävät järjestelmät perustuvat?

Yleisesti tutkimuksen tuloksista voidaan todeta, että mikropalveluiden kehittämisessä hyödynnettävien julkaisuprosessien ja toimintamallien soveltaminen oli hyvin linjassa aiemman tutkimuksen kanssa. Sama voidaan todeta myös mikropalveluarkkitehtuuriin perustuvassa kehittämisessä tärkeinä pidetyistä teknisistä mahdollistajista. Tuloksissa korostui erityisesti automaation ja valmiiden työkalujen merkitys, jotka ovat molemmat DevOps käytänteisiin liittyviä periaatteita. Lisäksi automaattisten julkaisukäytänteiden hyödyntämistä pidettiin välttämättömänä mikropalveluita kehitettäessä. Mikropalveluarkkitehtuuriin perustuvia järjestelmien kehittämiseen liittyen oleellisina ratkaisuina tunnistettiin:

- DDD-suunnitteluperiaatteiden seuraaminen
- Mikropalveluiden teknologiariippumattomuus
- Kehitysprosessin läpi jatkuva suunnittelu ja seuranta
- Mikropalveluiden välinen REST-periaatteiden mukainen kommunikointi
- Rajapintojen kautta välitettävien viestien muodon standardointi
- Mikropalveluiden rajattu pääsy tietovarastoihin
- Pilvialustojen hyödyntäminen mikropalveluarkkitehtuuriin hallinnoinnissa
- Automaation korostaminen julkaisemisessa ja monitoroinnissa

- Konttien hyödyntäminen mikropalveluiden julkaisemisessa

Seuraavissa alaluvuissa käsitellään tutkimuksen tulokset teemakohtaisesti ja pohditaan, miten tutkimuksen tulokset liittyvät aiempaan tutkimukseen. Viimeisessä alaluvussa myös esitellään potentiaalisia jatkotutkimusehdotuksia.

6.1 Julkaisuprosessit

Mikropalveluarkkitehtuuriin perustuvassa kehittämisessä erittäin merkittävänä pidettiin jatkuvan integraation ja jatkuvan toimittamisen periaatteita. Jatkuvan integraation ja jatkuvan toimittamisen periaatteita pidettiin edellytyksenä mikropalveluarkkitehtuurin tehokkaalle käytölle. Haastatteluissa ei kuitenkaan noussut esiin, että yhdelläkään haastateltavista olisi ollut kokemuksia jatkuvasta julkaisemisesta, missä jokainen muutos viedään suoraan tuotantoympäristöön. Tästä voidaan päätellä, että jatkuvaan julkaisemiseen liittyvät periaateet eivät ole ensisijainen tavoite mikropalvaleuarkkitehtuurissa. Julkaisuprosesseihin liittyvät löydökset olivat hyvin linjassa esimerkiksi Callanan ja Splilanen (2016) tutkimuksen kanssa, jossa painotettiin automaattisten julkaisuprosessien tärkeyttä erityisesti mikropalveluiden yhteydessä. Myös Soldanin ym. (2018) tutkimuksessa automaattisten julkaisukäytänteiden hyödyntämistä pidettiin erittäin tärkeänä mikropalveluarkkitehtuuriin perustuvia järjestelmiä rakennettaessa. Heidän mukaansa mikropalveluarkkitehtuuri mahdollistaa entistä tehokkaamman jatkuvan integraation ja jatkuvan toimittamisen. Pääasiallisina ajureina toimivat mikropalveluiden itsenäisyys ja niiden pieni koko, mikä mahdollistaa nopean muutoksiin vastaamisen. Mikropalveluiden modulaarisuutta ja itsenäisyyttä pidettiin erittäin merkittävä tekijänä, joka myös on tunnistettu tärkeänä mahdollistajana jatkuvia julkaisuprosesseja rakennettaessa (Ståhl ym., 2017).

Automatisoitujen julkaisuprosessien rakentamiseen liittyen konttiteknologioiden käyttö oli merkittävässä roolissa. Lisäksi oli havaittavissa, että synonyymina konteille käytettiin Dockeria, josta on muodostunut standardi tapa paketoida ohjelmistoja. Tämä oli havaittavissa myös konttiteknologioita käsittelevistä tutkimusartikkeleista (Jaramillo ym., 2016; Balalaie ym., 2018, Pahl ym., 2018). Myös Zimmermannin (2017) määritelmän mukaan konttiteknologiat ovat oleellinen osa mikropalveluarkkitehtuuria. Fransescon ym. (2017) mukaan konttien avulla suoritettava virtualisointi on merkittävä mahdollistaja mikropalveluarkkitehtuurissa. Konttien käytön merkitys korostui erityisesti mitä suuremmista järjestelmistä oli kyse. Konttien tärkeänä ominaisuutena pidettiin ohjelmakoodin eristämistä alustasta. Tämä ominaisuus näyttelee erittäin suurta roolia tilanteissa, joissa samaan järjestelmään kuuluvia mikropalveluita on päädytty kehittämään useilla eri ohjelmointikielillä. Konttien avulla pystytään välttämään jokaisen ohjelmointikielen vaatimien kirjastojen erillinen asentamisen ympäristöihin, joissa järjestelmää ajetaan.

Aiemman tutkimuskirjallisuuden sekä tämän työn tulosten perusteella voidaan todeta, että automaatioon perustuvat julkaisukäytänteet ovat merkittävässä roolissa mikropalveluarkkitehtuurin perustuvia järjestelmiä rakennettaessa.

6.2 Kehittämistä tukevat toimintamallit

Haastateltavien tärkeiksi kokemat toimintamallit liittyen mikropalveluiden kehittämiseen olivat hyvin linjassa esimerkiksi Jabbarin ym. (2018) ja Dyckin ym. (2015) suorittamien tutkimuksien kanssa, joissa pyrittiin määrittelemään DevOpsiin liittyviä periaatteita. Toistuvia teemoja olivat erityisesti automaatio, monitorointi ja yhteistyö. Myös Soldanin ym. (2018) mukaan DevOpsiin liittyvät käytänteet ovat ensiarvoisen tärkeitä mikropalveluita kehitettäessä. Haastatteluissa nousi esiin myös termi läpinäkyvyys, josta ei löytynyt mainintoja tämän työn kirjallisuuskatsauksessa käsitellystä materiaalista. Toisaalta läpinäkyvyys voidaan nähdä osana operoinnista ja kehittämisestä vastaavien sidosryhmien välistä yhteistyötä. Haastatteluissa korostui erityisesti valmiiden alustojen/työkalujen käyttö automaation saavuttamisessa. Smeds ym. (2015) tunnistivat DevOpsin yhdeksi keskeiseksi periaatteeksi valmiiden työkalujen hyödyntämisen. DevOpsin yhteydessä käytettävien työkalujen merkitystä korostavat myös Ebertin ym. (2016) ja Jamshidin ym. (2018) DevOps työkaluja esittelevät artikkelit. DevOpsiin liitettyjen työkalujen avulla pystytään toteuttamaan mikropalveluarkkitehtuurin kannalta useita tärkeitä toimintoja, mitkä liittyvät esimerkiksi mikropalveluiden julkaisemiseen, monitorointiin, skaalautumiseen ja virheiden käsittelemiseen (Balalaie ym., 2016).

Tästä voidaan päätellä, että mikropalveluarkkitehtuurin rakentaminen sisältää varsinaisen ohjelmiston kehittämisen lisäksi suuren määrän erilaisten konfiguraatioiden ja infrastruktuuriin liittyvien ratkaisuiden omaksumista. Tämä osaltaan tukee moniosaajista koostuvan DevOps-tiimin hyödyntämistä mikropalveluarkkitehtuuriin perustuvia järjestelmiä kehitettäessä. Tätä tukee myös Francescon ym. (2018) löydökset, joiden mukaan mikropalveluita tukevan infrastruktuurin rakentaminen koettiin yhtenä suurimpana haasteena vanhan järjestelmän muuntamisessa mikropalveluarkkitehtuuriin. Pahl ym. (2018) esittivät DevOpsin soveltuvan erityisesti pilvipohjaisten järjestelmien yhteyteen. Tämä on hyvin linjassa haastateltavien esittämien määritelmien mukana, joiden perusteella mikropalvelut ovat tyypillisesti pilviympäristöissä ajettavia järjestelmiä. Tämä vahvistaa osaltaan DevOpsin soveltuvuutta erityisesti mikropalveluarkkitehtuuriin keskittyvän kehittämisen yhteyteen.

Aiemman tutkimusmateriaalin ja myös tämän työn tulosten perusteella voidaan todeta, että DevOps-käytänteiden omaksumisella pystytään tehostamaan mikropalveluarkkitehtuurin kannalta kriittisen toimintojen toteuttamista

6.3 Mikropalveluiden muodostaminen

Lähes kaikissa mikropalveluiden kehittämistä käsittelevissä teemoissa tuli esiin mikropalveluiden itsenäisyyden säilyttämisen tärkeys, sillä ilman sitä koko arkkitehtuurityylin ydinidea häviää. Mikropalveluiden muodostamiseen liittyen haastatteluissa korostui erityisesti Evansin (2003) DDD-periaatteet. Esiin nousi myös muita merkittäviä tekijöitä, jotka pitää erityisesti huomioida jo mikropalveluiden suunnitteluvaiheessa. Näitä olivat mikropalveluiden skaalautumistarpeisiin perustuva jakaminen sekä usein muuttuvien osien eriyttäminen omiksi mikropalveluiksi. Lisäksi mikropalveluiden muodostamisessa nousi esiin mikropalveluiden koon merkitys. Liian suureksi suunniteltujen mikropalveluiden testaaminen ja kehittäminen hankaloituu. Tämä on myös vastoin haastatteluissa esiin noussutta ajatusta, minkä mukaan yhden mikropalvelun tulisi olla kooltaan sellainen, että yksi kehittäjä pystyy sisäistämään ja kehittämään mikropalvelua kokonaisuutena.

Haastatteluissa esiin nousseet ajatukset mikropalveluiden koosta ovat hyvin linjassa esimerkiksi Dragonin ym. (2016) esittämän määritelmän kanssa, minkä mukaan liian suureksi kasvaneet mikropalvelut aiheuttavat virheitä sekä niiden toiminnollisuuden ymmärtäminen hankaloituu. Haastattelujen perusteella voidaan päätellä, että mikropalveluarkkitehtuuriin perustuvan järjestelmän kehittäminen vaatii huomattavasti tarkempaa ja syvällisempää ymmärrystä aihealueesta verrattuna suurempiin kokonaisuuksiin perustuviin järjestelmiin. Lisäksi mikropalveluiden koon säilyttäminen ymmärrettävänä mahdollistaa mikropalveluarkkitehtuurin ketterän kehittämisen. Näin ollen voidaan todeta, että suunnitelmallisuuden merkitys kasvaa mikropalveluarkkitehtuuriin perustuvaa järjestelmää rakennettaessa. Myös Zimmermanin (2017) mukaan mikropalveluiden kasvanut suosio on johtanut arkkitehtuurin ja ennakkoon tapahtuvan suunnittelemisen korostumiseen, mikä on vastoin ketterän kehittämisen perusajatusta, minkä mukaan ennakkoon tapahtuva suunnittelu tulisi olla minimaalista. Tämä myös osaltaan tukee Zimmermannin (2017) ajatusta siitä, että mielenkiinto on siirtymässä ketteristä menetelmistä jatkuvan toimittamisen tavoittelemiseen sekä DevOps käytänteisiin.

6.4 Mikropalveluiden kehittäminen

Mikropalveluiden rakentamiseen liittyen haastatteluissa pidettiin potentiaalisena vaihtoehtona monoliittisen järjestelmän rakentamista, joka voitaisiin myöhemmässä vaiheessa jakaa mikropalveluiksi. Samankaltaisia ajatuksia on esittänyt Newman (2015, 249), jonka mukaan mikropalveluiden tunnistaminen on huomattavasti helpompaa sen jälkeen, kun kehitettävän järjestelmä vaatimukset ja niihin liittyvät haasteet on ymmärretty. Mikropalveluiden kehittämiseen liittyen esiin nousi myös mahdollisuus valita parhaiten tilanteeseen soveltuva ohjelmointikieli. Monikielisyys esitetään myös keskeisenä ominaisuutena mikro-

palveluarkkitehtuurin määritelmissä (Fowler & Lewis, 2014; Zimmermann, 2017). Haastatteluissa monikielisyyttä pidettiin hyvänä ominaisuutena, mutta siihen myös liitettiin haasteita, mitkä tulee huomioida ennen, kun päädytään soveltamaan useampaa kuin yhtä ohjelmointikieltä. Myös Soldani ym. (2018) tunnistivat teknologia riippumattomuuden yhdeksi mikropalveluarkkitehtuurin avulla saavutettavaksi hyödyiksi, joka mahdollistaa mikropalveluiden toteuttamisen parhaiten tilanteeseen soveltuvalla ohjelmointikielellä. Monikieliseen kehittämiseen päädyttäessä on tärkeää arvioida, miten suuri hyöty sillä saavutetaan verrattuna haasteisiin, kuten esimerkiksi uusien julkaisumekanismien rakentamiseen. Myös Chen (2018) esitti monikieliseen kehittämiseen liittyen samankaltaisia ajatuksia, kun haastateltavat. Hän myös korosti valittujen teknologioiden tarkkaa arvioimista ennen niiden käyttöönottoa. Useampaan ohjelmointikieleen perustuvia mikropalvelujärjestelmiä kehitettäessä korostuu myös erityisesti konttiteknologioiden käyttö, joiden avulla valitut ohjelmointikielet saadaan eriytettyä palvelinympäristöistä.

Mikropalveluarkkitehtuuriin liitettiin haastatteluiden perusteella monimutkaisuus ja haastavuus etenkin kehittäjille, jotka eivät ole työskennelleet aiemmin hajautettujen järjestelmien parissa. Haastatteluissa myös ilmeni kuinka olisi tärkeää, että kehitystiimissä olisi mukana henkilöitä, joilla on syvällinen ymmärrys mikropalveluarkkitehtuurista. Tärkeäksi koettiin myös se, että arkkitehtuuriin noudattamista seurataan aktiivisesti, sillä mikropalveluarkkitehtuurissa, virheiden tekeminen on huomattavasti helpompaa kuin suoraviivaisessa monoliittisessa arkkitehtuurissa. Francesco ym. (2018) esittivät samankaltaisia ajatuksia tutkimuksessaan. Mikropalveluiden poikkeuksellinen tapa kommunikoida ainoastaan verkon yli vaatii kehittäjiltä uudenlaisen ajatusmallin omaksumista. Rajapintojen kautta tapahtuvat kyselyt tulevat resurssien kannalta huomattavasti kalliimmiksi kuin suorat kyselyt tietokantaan. Tätä tukee myös Taibin ym. (2017) löydökset, joiden mukaan mikropalveluarkkitehtuuri on kehittäjille haastavampi sisäistää ja se lisää järjestelmän monimutkaisuutta.

Mikropalveluarkkitehtuuri tuo mukanaan paljon hyötyjä, jotka mahdollistavat järjestelmän tehokkaan kehittämisen, näiden hyötyjen saavuttaminen vaatii kutenkin kehittäjiltä uusien asioiden omaksumista. Mikropalveluiden itsenäisyyden vuoksi muutoksien tekeminen järjestelmään vaatii tarkempaa suunnittelua kuin ennen. Esimerkiksi verkon yli tapahtuva kommunikointi ja itsenäisten tietovarastointiratkaisuiden suunnitteleminen ovat tärkeitä osa-alueita, jotka tulee huomioida mikropalveluita kehitettäessä.

6.5 Mikropalveluiden välinen kommunikointi

Mikropalveluiden kommunikoinnista keskusteltaessa esiin nousi rajapintojen standardoinnin tärkeys. Standardoimalla järjestelmän sisäinen kommunikointi, voidaan välttyä virhealttiilta ja vaikeasti ymmärrettäviltä rajapinnoilta. Lisäksi esiin nousi rajapintojen yhteensopivuuden säilyttäminen, järjestelmän kehittyessä. Yleisesti kommunikointia pidettiin erittäin merkittävänä osa-alueena mik-

ropalveluarkkitehtuurissa, joka vaatii tarkkaa suunnittelua ja johon tulisi kiinnittää erityistä huomiota projektin alusta alkaen. Haastateltavien mukaan kommunikoinnissa tyypillisesti hyödynnettävät yhtenäinen rajapinta (Api Gateway) ja REST-periaatteet olivat hyvin linjassa aiemman kirjallisuuden kanssa (Newman, 2015, 50; Fransesco ym., 2017; Taibi ym., 2018; Soldani ym., 2018). Haastatteluissa nousi esiin myös tapahtumapohjaisen arkkitehtuuriin hyödyntäminen mikropalveluiden välisessä kommunikoinnissa. Myös Richardson (2018, 184) tunnistaa kirjassaan tämän suunnittelumallin potentiaaliseksi tavaksi rakentaa mikropalveluarkkitehtuuri. Tapahtumapohjaisen kommunikoinnin soveltamisesta mikropalveluarkkitehtuurin yhteydessä ei kuitenkaan tämän tutkimuksen puitteissa löytynyt aiempia tieteellisiä tutkimuksia. Haastatteluiden perusteella voidaan kuitenkin todeta, että tapahtumapohjaista kommunikointia pidetään sopivana kommunikointitapana mikropalveluarkkitehtuuriin pohjautuvissa järjestelmissä. Mikropalveluiden kommunikointiin liittyvissä keskusteluissa nousi myös esiin, kuinka palveluiden välinen kommunikointi on pidettävä minimaalisena ja yksinkertaisena. Näin voidaan tehostaa mikropalveluarkkitehtuurin toimintaa, sillä suuret verkon yli siirrettävät kuormat hidastavat järjestelmää, kuten myös liian tiheä kommunikointi, jossa tieto joudutaan koostamaan useista mikropalveluista. Tätä tukee myös Viggiaton ym. (2018) löydökset, joissa verkon yli tapahtuvat kutsut tunnistettiin mikropalveluarkkitehtuuria liittyväksi haasteeksi.

6.6 Datan hallinta

Data hallintaan liittyen haastatteluissa nousi esiin mahdollisuus usean eri tietokantatyypin soveltamisesta mikropalveluarkkitehtuurissa. Tätä pidettiin yhtenä mikropalveluarkkitehtuurin hyötynä, tulos on myös linjassa aiemman tutkimuksen kanssa. Soldani ym. (2018) tunnistivat mikropalveluarkkitehtuuriin yhdeksi keskeisistä hyödyistä eri tietokantajärjestelmien soveltamisen. Kirjallisuuskatsauksen perusteella mikropalveluarkkitehtuuriin liitettiin suunnittelumalli, jonka mukaan jokaisen tietovaraston tarvitsevan mikropalvelun tulisi omistaa oma tietokantansa (Taibi ym., 2017). Haastattelujen perusteella tämä olisi ihanteellinen tilanne, mutta todellisuudessa siihen on hyvin vaikea päästä esimerkiksi tilanteissa, missä vanhoja järjestelmiä ollaan modernisoimassa. Esiin nousi myös ajatus, jonka mukaan tietokantojen käyttö voitaisiin rajoittaa vain tiettyihin tauluihin siten, että yhteen tietokantaan olisi ainoastaan yksi kirjoittava mikropalvelu, mutta lukuoperaatiot sallittaisiin myös muille mikropalveluille. Jaetun tietokannan soveltaminen on kuitenkin tunnistettu yhdeksi mikropalveluarkkitehtuuriin liitetyksi antisuunnittelumalliksi, mitä tulisi välttää (Taibi & Lenardruzzi, 2018). Toisaalta tilanteissa, joissa ennalta tiedetään, että tietokantamalli tulee säilymään muuttomattona, jaetun tietokannan soveltaminen voi olla potentiaalinen vaihtoehto. Haastatteluissa nousi myös esiin eri tietokantavalintojen avulla saavutettava parempi suorituskyky. Useiden eri tietokantatyyppien soveltaminen on tunnistettu myös aiemmissa tutkimuksissa

oleelliseksi osaksi mikropalveluarkkitehtuuria (Viennot, Lécuyer, Bell, Geambasu, & Nieh, 2015; Ghofrani & Lübke, 2018).

Hajautettuja transaktioita pidettiin haastattelujen perusteella keskeisenä haasteena. Saman toteavat myös Viggiaton ym. (2018) kyselytutkimuksessaan, jonka mukaan hajautetut transaktiot olivat yksi mikropalveluarkkitehtuurin merkittävimpänä pidetyistä haasteista. Hajautettujen transaktioiden käsittelemiseksi haastatteluissa nousi kompensoivien transaktioiden hyödyntäminen, mitä myös on yleisesti ehdotettu aiemmassa tutkimuskirjallisuudessa hajautettujen transaktioiden käsittelemiseen. Haastatteluissa korostui myös, kuinka hajautettujen transaktioiden käsitteleminen olisi huomioitava jo suunnitteluvaiheessa, jonka jälkeen kaikkien mikropalveluiden on kyettävä käsittelemään hajautettuja transaktioita vaatimusten asettamalla tavalla. Tämä aiheuttaa erityisiä haasteita liittyen monikieliseen kehittämiseen, sillä transaktioiden käsittelemiseen on olemassa viitekehyksiä, mutta valmiiden viitekehyksien hyödyntäminen rajaa vaihtoehtoja, joilla itse mikropalvelu voidaan kirjoittaa. Haastatteluissa ilmeni, että hajautettuja transaktioita tulisi välttää suunnittelemalla järjestelmä siten, että niitä ei pääse syntymään. Samoilla linjoilla on esimerkiksi Newman (2015, 93), jonka mukaan mikropalveluissa päädytäänkin mieluummin ratkaisuihin, joissa hajautetut transaktiot suoritetaan lokaaleina transaktioina, jonka vuoksi mikropalveluarkkitehtuuri seuraa usein BASE-periaatteita.

Yllä käsitellyt asiat myös tukevat omalta osaltaan mikropalveluarkkitehtuuriin liitettyä suunnitelmallisuuden tärkeyttä. Eriytettyjen tietokantojen hyödyntämistä voidaan pitää merkittävänä osana mikropalveluarkkitehtuuria, millä on suuri merkitys mikropalveluiden itsenäisyyden säilyttämisessä.

6.7 Monitorointi ja virheiden käsitteleminen

Monitorointiin liittyen haastatteluissa nousi esiin keskitettyjen lokitietojen keräämisen merkitys sekä niiden pohjalta tapahtuva automaattinen virhetilanteista toipuminen sekä raportointi. Ratkaisuksi ehdotettiin mikropalveluarkkitehtuuriin toteuttamista konttien avulla, jolloin päästään hyötymään Kubernetesin kaltaisten konttien automaattiseen hallintaan tarkoitettujen työkalujen tarjoamista ominaisuuksista. Monitorointi ja virheiden käsitteleminen mikropalveluarkkitehtuurissa liitettiin automaatioon, joka omalta osaltaan vahvistaa DevOps-käytänteiden merkitystä mikropalveluarkkitehtuurin yhteydessä. Haastatteluissa monitorointiin ja virheiden käsittelyyn liitetyt ajatukset olivat hyvin linjassa esimerkiksi Jamshidin ym. (2018) esittämien ajatusten mukaan, jossa suurta roolia näyttelevät kontit sekä muut automaation mahdollistavat valmiit työkalut. Lokitietoihin keskittyvän monitoroinnin lisäksi erityisen tärkeänä haastatteluissa pidettiin järjestelmän kokonaistilan seurantaa, jonka avulla voidaan hallinnoida ja seurata järjestelmän toimintaa. Fransesco ym. (2017) tunnistavat järjestelmän tilan kokonaisvaltaisen seurannan tärkeäksi osaksi mikropalveluarkkitehtuuria. He myös totesivat sen korostavan mikropalveluarkkitehtuurin ja DevOpsin välistä riippuvuutta.

6.8 Skaalautuminen ja löydettävyys

Skaalautumiseen ja löydettävyyteen liittyvien toimintojen toteuttamisessa korostui valmiiden ratkaisuiden hyödyntäminen, jota myös tukevat Jamshidin ym. (2018) esittämät ajatukset mikropalveluarkkitehtuuriin liitetyistä työkaluista ja siitä mihin niitä sovelletaan. Skaalautuvuuteen liittyviä löydöksiä tukevat myös Giovannin ym. (2015) esittämät mikropalveluiden skaalautuvuuteen liittyvät kolme ratkaisua. Heidän mukaansa skaalautuminen voidaan hoitaa pilvipalvelutarjoajien alustaa hyödyntämällä, se voidaan rakentaa suoraan mikropalveluun hyödyntäen kolmansien osapuolien toteuttamia viitekehyksiä tai hyödyntäen kontteja sekä niiden hallinnointiin tarkoitettuja ohjelmistoja. Viimeinen vaihtoehto tarjoaa selkeästi eniten joustavuutta, eikä lukitse järjestelmää tiettyihin pilvipalvelutarjoajiin tai ohjelmointikehyksiin. Tämä osaltaan tukee haastelevien vastauksia, joissa mikropalveluiden skaalautuvuuden ratkaisemiksi ehdotettiin erityisesti Kubernetesin ja Dockerin käyttöä.

6.9 Jatkotutkimusehdotukset

Tässä työssä pyrittiin käsittelemään mikropalveluarkkitehtuuria kokonaisuutena, jonka vuoksi yhteenkään teemaan ei tämän työn laajuus huomioiden voitu paneutua kovin syvällisesti. Näin ollen potentiaalisina jatkotutkimus mahdollisuuksina voidaan pitää jokaisen tunnistetun teeman tarkastelua omana kokonaisuutena, joka mahdollistaisi teeman syvällisemmän analysoinnin. Tässä työssä tunnistettiin myös useita haasteita, mitkä liittyvät mikropalveluarkkitehtuuriin, näin ollen myös näiden syvällisempi tarkastelu olisi tulevaisuudessa hyödyllistä. Mielenkiintoista olisi myös selvittää DevOps-käytänteiden suosioita liittyen mikropalveluiden kehittämiseen, esimerkiksi kattavamman kyselytutkimuksen muodossa. Tämän työn perusteella DevOps-käytänteitä tarvitaan mikropalveluiden kehittämisessä, mutta termiä pidetään edelleen epämääräisenä. Teknisestä näkökulmasta potentiaalisina jatkotutkimusmahdollisuuksina voidaan nähdä erilaiset käytännön toteutukset liittyen tässä työssä tunnistettuihin teemoihin.

7 YHTEENVETO

Tässä työssä mikropalveluarkkitehtuuriin liitettyjä käytänteitä pyrittiin selvittämään laadullisten asiantuntijahaastatteluiden avulla. Tämän työn tuloksista voivat hyötyä erityisesti organisaatiot, jotka suunnittelevat mikropalveluarkkitehtuurin soveltamista. Tutkimuksen tulokset olivat hyvin linjassa kirjallisuuskatsauksen perusteella mikropalveluarkkitehtuurin kannalta oleellisiksi tunnustettujen tekijöiden kanssa. Yhteenvetona tutkimuksen tuloksista voidaan todeta, että merkittävää osaa mikropalveluarkkitehtuurissa näyttelee järjestelmän automaattiseen julkaisemiseen tähtäävän infrastruktuurin rakentaminen, mitä pidettiin välttämättömänä toimenpiteenä mikropalveluihin pohjautuvia järjestelmiä kehitettäessä. Tuloksista kävi myös ilmi, että omaksumalla DevOps-toimintamalliin liittyviä käytänteitä, voidaan tehostaa automaatioon tähtäävän infrastruktuurin rakentamista. Mikropalveluarkkitehtuurissa DevOps-käytänteistä korostui erityisesti yhteistyön, monitoroinnin ja automaation merkitys. Lisäksi useista mikropalveluista koostuvia järjestelmiä rakennettaessa kokonaisuuden merkitys korostuu, jolloin myös infrastruktuurin suunnittelemisen merkitys kasvaa. Mikropalveluarkkitehtuurin perustuvin järjestelmien infrastruktuurin hallinta on ottanut viime vuosina suuria harppauksia ja näin ollen esimerkiksi konttipohjaisten järjestelmien automaattiseen seurantaan ja skaalaamiseen on tarjolla useita ratkaisuja. Tämän tutkimuksen ja kirjallisuuskatsauksen perusteella mikropalveluarkkitehtuurin yhteydessä hyödynnetyistä työkaluista voidaan nostaa esiin erityisesti Kubernets ja Docker, joiden avulla kyetään automatisoimaan julkaisemiseen ja hallinnoitiin liittyviä prosesseja. Näin ollen mikropalveluiden kehittämisessä voidaan panostaa enemmän liiketoimintavaatimusten seurantaan sekä suurena haasteena pidettyyn oikean palvelujaon löytämiseen.

Tässä työssä mikropalveluiden suunnitteleminen tunnistettiin yhtenä haastavimmista osa-alueista koko mikropalveluarkkitehtuurissa, oikeanlaisen palvelujaon löytämiseksi ehdotettiin DDD-periaatteiden seuraamista. Mikropalveluiden rakentamisessa tärkeimpänä huomioitavana asiana voidaan pitää yksittäisten mikropalveluiden itsenäisyyden säilyttämistä. Mikropalveluiden itsenäisyydellä saavutetaan teknologia riippumattomuus, mikä mahdollistaa

palveluiden kehittämisen parhaiten tilanteeseen soveltuvilla teknologioilla. Mikropalveluiden merkittävänä arvolupauksena toimii palveluiden helppo skaalautuvuus. Vaikka skaalautuvuus voidaan nähdä alkuperäisenä syynä mikropalveluiden yleistymiseen, tämän työn perusteella niitä sovelletaan myös muiden ominaisuuksien vuoksi. Esimerkiksi mikropalveluiden itsenäinen kehittäminen ja julkaiseminen nähtiin tärkeinä ominaisuuksina. Mikropalveluiden välisessä kommunikoinnissa korostui standardisoinnin tärkeys sekä REST-periaatteiden mukainen kommunikointi. Lisäksi mikropalveluiden välinen kommunikointi tulee pyrkiä pitämään yksinkertaisena ja kevyenä. Tiedonvarastoinnin kannalta haastateltavilla ei ollut aivan yhtä jyrkkiä näkemyksiä, kuin kirjallisuuskatsauksen perusteella tyypillisesti mikropalveluarkkitehtuurin yhteydessä esitettiin. Tietokantojen eriyttämistä pidettiin kuitenkin tavoiteltavana tilana, jonka saavuttaminen aiheuttaa haasteita erityisesti vanhoja järjestelmiä modernisoitaessa. Monitorointiin, virheiden käsittelyyn ja skaalautumiseen liittyvien toimintojen toteuttamiseen sovelletaan tyypillisesti valmiita ratkaisuja. Näin toimimalla mikropalveluiden sisältämät toiminnollisuudet kyetään pitämään minimaalisina ja helposti ymmärrettävinä.

7.1 Tutkimukseen liittyvät rajoitteet ja menetelmien arviointi

Tässä työssä suoritettiin katsaus mikropalveluarkkitehtuuria tukeviin tekijöihin, kirjallisuuskatsauksen avulla pyrittiin tunnistamaan aiheen kannalta merkittävät teemat, joita käsiteltiin työn empiirisessä osassa. Aiheen laajuuden huomioiden teemoiksi olisi voinut valita myös erilaisia näkökulmia, kuten esimerkiksi tietoturvan, jota tässä työssä ei käsitelty lainkaan. Tutkimuksessa hyödynnetty lähdemateriaali oli myös varsin tuoretta, jonka vuoksi sen merkittävyyden arvioiminen oli haastavaa, vähäisten lainausten vuoksi. Tutkimuksen toteuttamisessa seurattiin alalle tyypilliseksi tutkimusmenetelmäksi tunnistettua laadullista haastattelumenetelmää. Haastattelujen toteuttamisessa käytetyt teemat perustuivat tutkimuskysymyksien kannalta merkittävien elementtien ympärille, joiden tunnistamiseen hyödynnettiin aiheeseen liittyvää merkittävää kirjallisuutta sekä aiempia tutkimuksia. Haastatteluihin valittiin teemahaastattelulle tyypillisesti henkilöitä, joilla tiedettiin jo etukäteen olevan runsaasti tietämystä mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämisestä. Tavoitteeksi asetettiin alun perin kymmenen haastattelua. Määrä kuitenkin päätettiin jättää seitsemään, sillä aineistossa oli havaittavissa selkää saturoitumista. Haastattelujen analysoinnissa käytetyt menetelmät olivat perusteltuja ja yleisesti laadullisen materiaalin analysoinnissa hyödynnettyjä.

7.2 Tulevaisuuden näkymät

Mikropalveluarkkitehtuuri on edelleen kehittyvässä tilassa oleva arkkitehtuurityyli, jonka ympärille rakentuu jatkuvasti uusia työkaluja, joiden avulla pystytään tehostamaan mikropalveluiden kehittämistä, julkaisemista ja ylläpitoa (Francesco, Lago, & Malavolta, 2019). Myös tämän työ tuloksista voidaan havaita, että mikropalveluarkkitehtuuri on saavuttanut tilan, jossa sen soveltamiseen liittyen voidaan tunnistaa jo useita parhaita käytänteitä. Tämän pitäisi entisestään tukeva ja helpottaa organisaatioiden päätöstä soveltaa mikropalveluarkkitehtuuria. Samalla on kuitenkin tärkeää muistaa, että mikropalveluarkkitehtuuri ei kuitenkaan välttämättä sovellu kaikkiin tilanteisiin. Näin ollen sen soveltamista tulisikin aina arvioida kehitettävä järjestelmän vaatimuksien pohjalta, eikä pelkästään mikropalveluiden kautta saavutettavien potentiaalisten hyötyjen kautta. Mikropalveluarkkitehtuurin hyödyntäminen tulee tulevaisuudessa todennäköisesti yleistymään, kun vanhoja järjestelmiä modernisoidaan ja tai korvataan kokonaan uusilla. Mikropalveluarkkitehtuurin yleistymisen myötä myös DevOps-käytänteiden soveltaminen tulee yleistymään entisestään.

LÄHTEET

Abbott, M. L. & Fisher, M. T. (2009). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise.* Addison-Wesley.

Adams, B. & McIntosh, S. (2016). Modern Release Engineering in a Nutshell - Why Researchers Should Care. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, (s. 79–90). Suita.

Alshuqayran, N., Ali, N. & Evans, R. (2016). A Systematic Mapping Study in Microservices Architecture. *IEEE 9th International Conference on Service-Oriented Computing and Application* (s. 44-51). Macau: IEEE.

Auerbach, C. F. & Silverstein, L. B. (2003). *Qualitative Data: An Introduction to Coding and Analysis.* New York: New York University Press.

Balalaie, A., Heydarnoori , A. & Jamshidi , P. (2016). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In A. Celesti & P. Leitner, *(eds) Advances in Service-Oriented and Cloud Computing. ESOCC 2015. Communications in Computer and Information Science* (s. 201–216). Cham: Springer.

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

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A. & Lynn, T. (2018). Microservices migration patterns. *Software Practice and Experience 48(3)*, 1–24.

Beck, K. (1999). *Extreme programming explained: embrace change.* Boston: Addison-Wesley Longman Publishing Co.

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 81–84.

Bucchiarone, A., Mazzara, M., Dustdar, S., Dragoni, N. & Larsen, S. T. (2018). From Monolithic to Microservices An Experience Report from the Banking Domain. *IEEE Software*, 50–55.

Callanan, M. & Splilane, A. (2016). DevOps Making It Easy to Do the Right Thing. *IEEE Software 33(3)*, 53–59.

Carcia-Molina, H. & Salem, K. (1987). Sagas. *Proceeding SIGMOD '87 Proceedings of the 1987 ACM SIGMOD international conference on Management of data* (s. 249–259). New York: ACM.

Cerny, T., Donahoo, M. j. & Trnka, M. T. (2017). Contextual understanding of microservice architecture. *ACM SIGAPP Applied Computing Review 17 (4)*, 29-45.

Chen, L. (2018). Microservices: Architecing for Continuos Delivery and DevOps. *2018 IEEE International Conference on Software Architecture (ICSA)* (s. 39–397). Seattle: IEEE.

Claps, G. G., Svensson, R. B. & Aurum, A. (2015). On the journey to continuous deployment: Technical and social chanllenges along the way. *Information and Software Technology*, 21–31.

Clark, K. J. (2016). *Microservices, SOA, and APIs: Friends or enemies? A comparison of key integration and application architecture concepts for an evolving enterprise.* developerWorks IBM.

Conway, M. E. (1968). How do committees invent. *Datamation, 14(4)*, 28–31.

De La Torre, C., Wagner, B. & Rousos, M. (2019). *.NET Microservices: Architecture for Containerizer .NET Applications.* Redmond: Microsoft Developer Division.

Dingsøyr, T. & Lassenius, C. (2016). Emerging themes in agile software development: Introduction to the special section on continuous value delivery. *Information and Software Technology*, 56–60.

Dingsøyra, T., Nerur, S. & Moe, N. B. (2012). A decade of agile methodologies: Towards explaining agile software development. *The Journal of Systems and Software*, 1213–1221.

Docker (2018). *Docker overview.* Haettu 8.12.2018 osoitteesta https://docs.docker.com/engine/docker-overview/

Dragoni, N., Giallorenzo, S., Llunch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017). Microservices: Yesterday, Today, and Tommorow. In M. Mazzara & B. Meyer, *Present and Ulterior Software Engineering* (s. 195-216). Cham: Springer.

Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R. & Safina, L. (2018). Microservices: How To Make Your Application Scale. In A. Petrenko & A. Voronkov, *Perspectives of System Informatics. PSI 2017* (s. 95–104). Cham: Springer.

Dyck, A., Penners, R. & Lichter, H. (2015). Towards Definitions for Release Engineering and DevOps. *2015 IEEE/ACM 3rd International Workshop on Release Engineering* (s. 3–3). Florence: IEEE.

Ebert, C., Gallardo, G., Hernantes, J. & Serrano, N. (2016). DevOps. *IEEE Software 33(3)*, 94–100.

Elberzhager, F., Arif, T., Naab, M., Süß, I. & Koban, S. (2017). From Agile Development to DevOps: Going Towards Faster Releases at High Quality – Experiences from an Industrial Context. *Software Quality Complexity and Challenges of Software Engineering in Emerging Technologies* (s. 33–44). Cham: Springer.

Endrei, M., Ang, J., Arsnjani, A., Chua, S., Comte, P., Krogdahl, P., . . . Newling, T. (2004). *Patterns: Service-Oriented Architecture and Web Services.* IBM Corporation.

Erich, F. M., Armit, C. & Daneva, M. (2017). A qualitative study of DevOps usage in practice F. *Journal of Software Evolution and Process*, 1–20.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addision Wesley.

Feitelson, D. G., Frachtenberg, E. & Beck, K. L. (2013). Development and Deployment at Facebook. *IEEE Internet Computing 17(4)*, 8–17.

Forsgren, N., Humble, J. & Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* IT Revolution Press.

Fowler, M. & Lewis, J. (2014). *Microservices: a definition of this new architectural term*. Haettu 15.12.2018 osoitteesta https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture

Francesco, P. D., Lago, P. & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 77–97.

Francesco, P., Lago, P. & Malavolta, I. (2018). Migrating Towards Microservice Architectures: An Industrial Survey. *2018 IEEE International Conference on Software Architecture (ICSA)* (s. 29–2909). Seattle: IEEE.

Francesco, P., Malavolta , I. & Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. *2017 IEEE International Conference on Software Architecture (ICSA)* (s. 21–30). Gothenburg: IEEE.

Ghofrani, J. & Lübke, D. (2018). Challenges of Microservices Architecture: A Survey on the State of the Practice. *Proceedings of the 10th Central European Workshop on Services and their Composition*, (s. 1–9). Dresden.

Giovanni, T., Brunner, S., Blöchlinger, M., Dudouet, F. & Edmonds, F. (2015). An architecture for self-managing microservices. *AIMC '15 Proceedings of the 1st International Workshop on Automated Incident Management in Cloud* (s. 19–24). Bordeaux: ACM.

Goldsmith, K. (2015, 12 3). *Microservices at Spotify.* Haettu 17.12.2018 osoitteesta https://www.kevingoldsmith.com/talks/microservices-at-spotify.html

Gysel, M., Kölbener, L., Giersche, W. & Zimmermann, O. (2016). Service Cutter: A Systematic Approach to Service Decomposition. In M. Aiello, E. Johanssen, S. Dustdar & I. Gerogievski, *Service-Oriented and Cloud Computing* (s. 185–200). Cham: Springer.

Hirsjärvi, S. & Hurme, H. (2000). *Tutkimushaastattelu Teemahaastattelun teoria ja käytäntö.* Helsinki: Yliopistopaino.

Hirsjärvi, S., Remes, P. & Sajavaara, P. (2009). *Tutki ja kirjoita (15. uudistettu painos).* Helsinki: Tammi.

Humble, J. & Farley, D. (2011). *Continuous delivery : reliable software releases through build, test, and deployment automation.* Upper Saddle River, NJ: Addison-Wesley.

Hüttermann, M. (2012). *DevOps for Developers.* Berkeley, CA: Apress.

Jabbari, R., Ali, N. B., Petersen, K. & Tanveer, B. (2016). What is DevOps? A Systematic Mapping Study on Definitions and Practices. *XP '16 Workshops Proceedings of the Scientific Workshop Proceedings of XP2016* (s. 1–11). New York: ACM.

Jabbari, R., Nauman, B., Petersen, A. & Tanveer, B. (2018). Towards a benefits dependency network for DevOps based on a systematic literature review. *Journal of Software: Evolution and Process 30(11)*, 1–26.

Jamshidi, P. & Pahl, C. (2016). Microservices: A Systematic Mapping Study. *In Proceedings of the 6th International Conference on Cloud Computing and Services Science*, (s. 137–146).

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J. & Tilkov, S. (2018). Microservices The Journey So Far and Challenges Ahead. *IEEE Software 35(3)*, 24–35.

Jaramillo, D., Nguyen, D. V. & Smart, R. (2016). Leveraging microservices architecture by using Docker technology. *SoutheastCon 2016* (s. 1–5). Norfolk: IEEE.

Kang, H., Le, M. & Tao, S. (2016). Container and Microservice Driven Design for Cloud Infrastructure DevOps. *2016 IEEE International Conference on Cloud Engineering (IC2E)* (s. 202–211). Berlin: IEEE.

Kaplan, B. & Maxwell, J. A. (2005). Qualitative Research Methods for Evaluating Computer Information Systems. In J. G. Anderson & C. E. Aydin, *Evaluating the Organizational Impact of Healthcare Information Systems. Health Informatics* (s. 30–55). New York, NY: Springer.

Kleppmann, M. (2017). *Designing data-intensive applications : the big ideas behind reliable, scalable, and maintainable systems.* O'Reilly: Sebastopol.

Kookarinrat, P. & Temtanapat, Y. (2016). Design and implementation of a decentralized message bus for microservices. *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)* (s. 1–6). Khon Kaen: IEEE.

Laukkanen, E., Itkonen, J. & Lassenius, C. (2017). Problems , causes and solutions when adopting continuous delivery — A systematic literature review. *Information and Software Technology*, 55–79.

Lwakatare, L. E., Kuvaja, P. & Oivo, M. (2015). Dimensions of DevOps. In C. Lasenius , D. Torgeir & P. Maria, *Agile Processes, in Software Engineering, and Extreme Programming* (s. 212–217). Cham: Springer.

Martin, A., Raponi, S., Combe, T. & Di Pietro, R. (2018). Dcoker ecosystem - Vulnerability Analysis. *Computer Communications 122* , 30–43.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* (239) 2, 76–90.

Millet, S. & Tune, N. (2015). *Patterns, Principles and Practices of Domain-Driven Design.* Indianapolis: John Wiley & Sons.

Molesky, J. & Humble, J. (2011). Why Enterprises Must Adopt Devops to Enable Continuous Delivery. *Cutter IT Journal*, 6–12.

Montesi, F. & Weber, J. (2016, 9 21). *Circuit Breakers, Discovery, and API Gateways in Microservices.* https://arxiv.org/abs/1609.05830v2

Myers, M. D. & Avison, D. (2002). An Introduction to Qualitative Research in Information Systems. In M. D. Myers & D. Avison, *Qualitative Research in Information Systems* (s. 3–12). London: Sage publications.

Myers, M. M. & Newman, M. (2007). The qualitative interview in IS research: Examing the craft. *Information and Organization 17 (*, 2–26.

Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M. (2016). *Microservice Architecture: ALIGNING PRINCIPLES, PRACTICES, AND CULTURE.* Sebastopol: O'Reilly.

Namiot, D. & Sneps-Sneppe, M. (2014). On Micro-services Architecture. *International Journal of Open Information Technologies*.

Newman, S. (2015). *Building Microservices.* Sebastopol: O'Reilly.

Nygard, M. T. (2007). *Release It! Desing and Deploy Production-Ready Software.* Dallas: Pragmatic Bookshelf.

O'Connor, R. V., Elger, P. & Clarke, P. M. (2017). Continuous software engineering—A microservices architecture perspective. *Journal of Software: Evolution and Process 29(11)*, 1–12.

Pahl, C. & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. *6th International Conference on Cloud Computing and Services Science* (s. 137-146). Rome: SCITEPRESS - Science and Technology Publications, Lda .

Pahl, C., Jamshidi, P. & Zimmermann, O. (2018). Architectural Principles for Cloud Software Architectural Principles for Cloud Software. *ACM Transactions on Internet Technology 18 (2)*, 1–21.

Perry, D. E. & Wolf, A. L. (1992). Foundations for the Study of Software Architecture. *Software Engineering Notes 17(4)*, 40–52.

Peshkin, A. (1993). The Goodness of Qualitative Reserach. *Educational Research (22) 2*, 23–29.

Pritchett, D. (2008). BASE: An Acid Alternative. *Queue - Object-Relational Mapping*, 48–55.

Puppet & Splunk. (2018). *Puppet.* Haettu 09.01.2019 osoitteesta https://puppet.com/resources/whitepaper/state-of-devops-report

Qualitative Research in Information Systems. (1997). *MIS Quarterly*, 241-242.

Rademacher, F., Sorgalla, J. & Sachweh, S. (2018). Challenges of Doamin-Driven Microservice Design: A Model-Driven Perspective. *IEEE Software 35 (3)*, 36–43.

Richards, M. (2015). *Microservices vs Servicer-Oriented Architecture.* Sebastopol: O'Reilly.

Richardson, C. (2018). *Microservice Patterns.* Shelter Island: Manning.

Riungu-Kalliosaari, L., Mäkinen, S., Lwakatare, L. E., Tiihonen, J. & Männistö, T. (2016). DevOps Adoption Benefits and Challenges in Practice. In P. Abramsson, A. Jedlitschka, A. N. Duc, M. Felderer, S. Amasaki & T. Mikkonen, *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016 Trondheim, Norway, November 22–24, 2016 Proceedings* (s. 590–597). Cham: Springer.

Rodríguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., . . . Oivo, M. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *The Journal of Systems and Software (123)*, 263–291.

Senapathi, M.;Buchan, J.;& Osman, H. (2018). DevOps Capabilities, Practices, and Challenges: Insights from a Case Study. *In Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018 (EASE'18)* (ss. 57–67). ACM: New York.

Shahin, M., Babar, M. A. & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access 5*, 3909-3943.

Sill, A. (2016). The Design and Architecture of Microservices. *IEEE Cloud Computing*, 76–80.

Smeds, J., Nybom, K. & Porres, I. (2015). DevOps: Definition and Perceived Adaption Impediments. In C. Lassenius, T. Dingsoyr & M. Paasivaara, *Agile Processes, in Software Engineering and Extreme Proramming* (s. 166–176). Cham: Springer.

Soldani, J., Tamburri, D. A. & Heuvel, W.-J. V. (2018). The pains and gains of microservices: A Systematic grey literature review. *The Journal of Systems and Software*, 215–232.

Ståhl, D. & Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *The Journal of Systems and Software*, 49–59.

Ståhl, D., Mårtensson, T. & Bosch, J. (2017). The continuity of continuous integration: Correlations and consequences. *The Journal of Systems and Software*, 150–167.

Sysdig. (2018, 29 toukokuu). *Sysdig*. Haeuttu 20.01.2019 osoitteesta https://sysdig.com/blog/2018-docker-usage-report/

Taibi, D. & Lenardruzzi, V. (2018). On the Definition of Microservice Bad Smells. *IEEE Software 35 (3)*, 56–62.

Taibi, D., Lenarduzzi, V. & Janes, A. (2017). *Microservices in Agile Software Development: a Workshop Based Study into Issues, Advantages, and Disadvantages.* Cologne: Association for Computing Machinery. ACM.

Taibi, D., Lenarduzzi, V. & Pahl, C. (2018). Architectural Patterns for Microservices: A Systematic Mapping Study. *Proceedings of the 8th International Conference on Cloud Computing and Services Science* (s. 221–232). SciTePress.

Tuomi, J. & Sarajärvi, A. (2010). Laadullinen tutkimus ja sisällönanalyysi. Helsinki: Tammi.

Viennot, N., Lécuyer, M., Bell, J., Geambasu, R. & Nieh, J. (2015). Synapse: a microservices architecture for heterogeneous-database web applications. *EuroSys '15 Proceedings of the Tenth European Conference on Computer Systems.* New York: ACM.

Viggiato, M., Terra, R., Rocha, H. & Valente, M. T. (2018). Microservices in Practice: A Survey Study. *6th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*, (s. 1–8). São Carlos.

Zhu, L., Bass, L. & Champlin-Scharff,, G. (2016). DevOps and Its Practices. *IEEE SOFTWARE 33(3)*, 32–34.

Zimmermann, O. (2017). Microservices tenets Agile approach to service development and deployment. *Computer Science - Research and Development 32(3-4)*, 301–310.

LIITE 1 HAASTATTELURUNKO

Esittely

- Olen tietojärjestelmätieteen opiskelija Jyväskylän yliopistosta. Tässä gradu tutkielmassa tavoitteena on selvittää mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämiseen liittyviä parhaita käytänteitä.

- Teemahaastattelu on muodoltaan ennalta määriteltyjen teemojen ympärille rakennettu vapaamuotoinen keskustelu, jonka ideana on keskustella kokemuksistasi liittyen mikropalveluarkkitehtuurin hyödyntämiseen. Voit myös vapaasti tuoda esiin teemojen ulkopuolisia aiheita. mitä pidetään tärkeänä ja mitä haastavana jne.

- Haastattelujen avulla kerättyjä tietoja käsitellään tutkielmassa anonyymisti ja ainoat tiedot ovat: työkokemus ja tehtävänimike sekä kokemus mikropalveluarkkitehtuurin parissa työskentelystä. Haastattelu jakaantuu kahteen pääteemaan ja kestää yhteensä noin 30–40 minuuttia.

Alkutiedot

- Kuinka monta vuotta henkilöllä on kokemusta ohjelmistokehittämisestä?
- Kuinka monta vuotta kokemusta mikropalveluarkkitehtuurista?
- Millaisessa tehtävässä toimit tällä hetkellä?

Määritelmä

- Miten määrittelet termin mikropalvelu?

Mikropalveluiden julkaisuprosessi

- Kehityksen aikana tapahtuva julkaiseminen sekä valmiin järjestelmän julkaiseminen. CI ja CD -merkitys
- Automaation merkitys.
- Kottien merkitys
- Julkaisualustojen hyödyntäminen, konttien käyttö.

- Saavutetaanko mikropalveluarkkitehtuurilla kokemuksesi mukaan merkittäviä hyötyjä järjestelmien julkaisemiseen liittyen.

Kehittämisessä hyödynnettävät toimintamallit

- Tiimin koostumus.
- Julkaisuprosessissa käytettävän infran rakentaminen?
- Miten voidaan helpottaa monimutkaiseen arkkitehtuuriin perustuvan järjestelmän kehittämistä?

Mikropalveluiden mudostaminen

- Miten mikropaveluiden jakaminen suoritetaan mikropalveluarkkitehtuuria rakennettaessa?

Kehittäminen

- Kokemuksesi mukaan nopeuttaako mikropalveluarkkitehtuuri kehittämistä? (useita itsenäisiä palveluita voitaisiin kehittää periaatteessa rinnakkain)
- Useiden eri teknologioiden hyödyntäminen samaan järjestelmään kuuluvissa mikropalveluissa?
- Lisääkö merkittävästi järjestelmän monimutkaisuutta?

Mikropalveluiden välinen kommunikointi

- Mitä ovat tyypilliset tiedonvälittämiseen käytettävät formaatit ja kommunikointi tavat?
- Mitä erityistä pitää kommunikoinnissa huomioida, kun rakennetaan mikropalveluarkkitehtuuriin perustuvaan järjestelmää?
- Mitä suunnittelumalleja kommunikoinnissa seurataan?
- Mikropalveluiden tilattomuus.

Datan käsitteleminen

- Miten mikropalveluiden pitäsi kommunikoida tietokantojen kanssa?
- Useiden eri tietokantateknologioiden hyödyntäminen

Mikropalveluiden löydettävyys

- Miten tämä ongelman tyypillisesti ratkaistaan?

Monitorointi ja virheiden käsittely

- Miksi monitoroinnin ja virheiden käsittely korostuu mikropalveluarkkitehtuurissa?
- Miten erityispiirteitä liittyy mikropalveluarkkitehtuurissa tapahtuvaan monitorointiin ja virheiden käsittelyyn?

Skaalautuminen

- Mitkä ovat tärkeimmät seikat, jotka pitää mikropalveluarkkitehtuurissa huomioida, jotta järjestelmän helppo skaalautuminen olisi mahdollista?
- Miten skaalautuminen käytännössä toteutetaan?