Jari Pennanen

Mikropalveluarkkitehtuurin hyödyt ja haitat

Mikropalveluilla toteutetun tietokantapohjaisen rajapintapalvelun vertailu monoliittiseen toteutukseen

Tietotekniikan kandidaatintutkielma

14. toukokuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Jari Pennanen

Yhteystiedot: `jari.o.o.pennanen@student.jyu.fi`

Ohjaaja: Antti-Juhani Kaijanaho

Työn nimi: Mikropalveluarkkitehtuurin hyödyt ja haitat: Mikropalveluilla toteutetun tietokantapohjaisen rajapintapalvelun vertailu monoliittiseen toteutukseen

Title in English: Pros and cons of microservice architecture: Comparing microservice implementation of database-backed API-service to monolithic implementation

Työ: Kandidaatintutkielma

Sivumäärä: 33+0

Tiivistelmä: Tässä kandidaatintutkielmassa käsitellään kirjallisuuskartoituksen keinoin mikropalveluarkkitehtuurin hyötyjä ja haittoja verrattuna monoliittiseen ohjelmistoon. Tarkastelussa on tietokantapohjaiset rajapintapalvelut, kuten HTTP:n päällä toimivat rajapinnat, jotka tallentavat tai hakevat tietokannasta tietoa. Kartoituksen tuloksena näyttäisi, että mikropalveluarkkitehtuurin hyödyt ovat saavutettavissa ainakin isommille organisaatioille, joilla on kykyä hallita hajautetun järjestelmän kompleksisuutta.

Avainsanat: mikropalvelu, mikropalveluarkkitehtuuri, monoliittinen ohjelmisto, tietokantapohjainen rajapintapalvelu

Abstract: This bachelor's thesis uses means of mapping study to investigate the pros and cons of microservice architecture compared to monolithic software. Under closer investigation is database-backed API-services such as HTTP-services which store and retrieve data from a database. Results of this mapping study indicate that pros of microservice architecture may be achieved at least by larger organizations which have a capability to manage the complexity of distributed systems.

Keywords: microservice, microservice architecture, monolithic application, database-backed API-service

Kuviot

Kuvio 1. Arkkitehtuurista riippumaton monoliittisen tietokantapohjaisen rajapintapalvelun yleiskuva. 5
Kuvio 2. Esimerkki mikropalvelujakoihin soveltuvista rajatuista konteksteista. . . . 8
Kuvio 3. Monoliittisen ja mikropalveluarkkitehtuurin vertaileminen sijoittelukaavion avulla. 13
Kuvio 4. Esimerkki monoliittisen ja mikropalveluarkkitehtuurin skaalautuvuudesta komponentin ja mikropalvelun C suhteen. 17
Kuvio 5. Monoliittisen ja mikropalveluarkkitehtuurin organisaatiorakenteet. 24

Sisältö

1 JOHDANTO 1

2 MONOLIITTISET OHJELMISTOT 3
2.1 Modulaarisuus ohjelmakooditasolla 3
2.2 Monoliittiset tietokantapohjaiset rajapintapalvelut 4

3 MIKROPALVELUARKKITEHTUURIN ESITTELY 6
3.1 Modulaarisuus jakamalla mikropalveluihin 6
3.2 Tietokanta kutakin mikropalvelua kohden 9
3.3 Mikropalvelujen yhdistäminen ja integrointimenetelmät 9
3.4 Ulkoisen rajapinnan toteutus 10
3.5 Mikropalveluarkkitehtuurin vertailu monoliittiseen 11

4 HYÖTYJEN JA HAITTOJEN VERTAILU 14
4.1 Teknologiset hyödyt ja haitat 14
4.1.1 Mikropalvelujakojen hankaluudet haittana 14
4.1.2 Mikropalvelujen uudelleenkäyttö ja päivitettävyys hyötynä 15
4.1.3 Skaalattavuuden hyödyt ja saatavuuden parantuminen 16
4.1.4 Tietoturvahaasteet haittana 18
4.1.5 Tietokannan hajautumisen haitat 19
4.2 Ohjelmistotuotannolliset hyödyt ja haitat 20
4.2.1 Automaation lisääntyminen ja aloituskustannusten nousu 20
4.2.2 Organisaatiorakenteen skaalautuvuus hyötynä 22

5 YHTEENVETO 25

LÄHTEET 26

1 Johdanto

Tämä tutkielma pyrkii vastaamaan kirjallisuuskartoituksen keinoin tutkimuskysymykseen, mitkä ovat mikropalveluarkkitehtuurilla toteutetun tietokantapohjaisen rajapintapalvelun hyödyt ja haitat verrattuna monoliittiseen toteutukseen. Tutkielmasta selviää myös monoliittisen ja mikropalveluarkkitehtuurin peruspiirteet. Monoliittinen tietokantapohjainen rajapintapalvelu otettiin hyötyjen ja haittojen arvioinnissa vertailukohdaksi, koska se on perinteisempi ohjelmistosuunnittelun näkökulma ja monet ohjelmistot ovat edelleen monoliittisia.

Aihe on ajankohtainen, sillä Taibin, Lenarduzzin ja Pahlin (2017) mukaan mikropalveluiden käyttö on lisääntynyt viime vuosina niin pienten kuin suurten yritysten joukossa. Lisäksi Taibin, Lenarduzzin ja Pahlin (2017) mukaan on vielä useita yrityksiä, jotka ovat vastahakoisia siirtymään mikropalveluihin, koska mikropalveluarkkitehtuurin hyödyistä ja haitoista ei ole tarpeeksi tietoa. Yritysten harkintaa ei helpota sekään, että käsitteenä mikropalvelu on Pautasson ym. (2017) mukaan vielä ristiriitainen, ja sille on kirjavasti määritelmiä. Ajankohtaisuuden ja ristiriitaisuuden takia aihe soveltuu kirjallisuuskartoitukseksi ja on hyödyllinen tapa kerätä tutkimuskysymyksen ympärillä olevat ristiriitaiset ja yhteneväiset tiedot.

Tietokantapohjainen rajapintapalvelu on ohjelmisto, joka tuottaa toimintonsa rajapintana, esimerkiksi HTTP:n päällä toimivana REST-, gRPC- tai GraphQL-rajapintana. Rajapintapalveluun voi olla yhdistyneenä esimerkiksi käyttöliittymä tai muita ulkoisia palveluita, mutta ne eivät ole osa rajapintaa. Lisäksi tietokantapohjaisen ohjelmiston tavoin se tallentaa ja hakee tietoja tuotetun rajapinnan kautta. Tutkielmassa pohditaan mikropalveluarkkitehtuuria erityisesti tietokantapohjaisen ohjelmiston näkökulmasta, mutta rajapinta otetaan huomioon kokonaisuutta vertailtaessa.

Lewis ja Fowler (2014) antavat yhden vanhimmista mikropalveluiden määritelmistä verkkoartikkelissaan, jossa mainitaan, että käsitteestä keskusteltiin ensimmäisen kerran vuonna 2011 käydyssä ohjelmistoarkkitehtuurityöpajassa. Tärkeämpä-

nä isompana pohjateoksena mikropalveluista on Newmannin kirjoittama kirja *Building microservices: designing fine-grained systems* (2015), johon useat artikkelit viittaavat. Toisaalta mikropalveluiden käsitettä kriittisesti tarkastellut Zimmermann (2017) summaa, että mikropalvelut eivät ole uusi käsite, vaan vanhemman palvelusuuntautuneen arkkitehtuurin (*Service-oriented architecture*) eräs toteutustapa.

Käsitteen synty voidaan nähdä myös osana teknologisia aaltoja, jotka liittyvät palvelusuuntautuneissa arkkitehtuureissa ja mikropalveluissa käytettyihin työkaluihin. Jamshidin ym. (2018) mukaan näitä teknologisia aaltoja, jotka edeltävät Lewisin ja Fowlerin (2014) määritelmää ja mahdollistivat mikropalvelujen työkalutuen, ovat mm. kontit (*Containers*), palvelulöytöohjelmistot (*Service discovery*), konttien orkestrointiohjelmistot (*Cointainer orchestration*) ja vikasietoiset kommunikointiohjelmistot (*Fault-tolerant communication*).

Lewisin ja Fowlerin (2014) mukaan mikropalvelu on ohjelmiston itsenäinen osa, joka suoritetaan omassa prosessissaan ja se keskustelee kevyiden mekanismien, kuten HTTP-resurssirajapinnan avulla. Keskeistä on, että kukin mikropalvelu on itsenäinen osa, joka mahdollistaa skaalattavuuden ja teknologiariippumattomuuden ohjelmiston osien välillä (Lewis ja Fowler 2014). Esimerkiksi voidaan valita eri ohjelmointikieli tai tietokantaohjelmisto kullekin mikropalvelulle soveltuvaksi.

Tämä tutkielma jakautuu viiteen päälukuun johdannon lisäksi. Luvussa 2 kuvataan, minkälainen on vertailukohtana käytetty monoliittinen tietokantapohjainen rajapintapalvelu. Luvussa 3 esitellään mikropalveluarkkitehtuuria ja sen keskeisiä piirteitä sekä verrataan sitä monoliittiseen ohjelmistoon. Luvussa 4 käsitellään tutkimuskysymyksenä olevat hyödyt ja haitat vertailemalla monoliittiseen arkkitehtuuriin. Luvussa 5 on yhteenveto ja johtopäätökset.

2 Monoliittiset ohjelmistot

Monoliittisella ohjelmistolla tarkoitetaan Dragonin ym. (2017) mukaan samaan ohjelmakoodipohjaan tukeutuvaa ohjelmistoa, jonka osia ei voida suorittaa itsenäisesti. Lisäksi Dragonin ym. (2017) mukaan monoliittinen ohjelmisto jakaa samat palvelinresurssit, kuten tietokannat, muistialueen ja tiedostot. Tässä luvussa kuvataan lyhyesti, mistä monoliittisissa ohjelmistoissa modulaarisuus muodostuu, sekä monoliittisen tietokantapohjaisen rajapintapalvelun arkkitehtuuri yleisellä tasolla.

2.1 Modulaarisuus ohjelmakooditasolla

Vaikka monoliittinen ohjelmisto tukeutuu samaan ohjelmakoodipohjaan, se voi olla silti modulaarinen. Martin (2017, s. 176) huomioi, että monoliittisessa ohjelmistossa voi olla ohjelmakooditasolla olevaa itsenäisyyttä (*Source-level decoupling*). Tämä tarkoittaa, että monoliittinen ohjelma voidaan jakaa uudelleenkäytettäviin komponentteihin ja moduuleihin, joiden keräämät toiminnot liittyvät oleellisesti toisiinsa.

Moduulilla tarkoitetaan Martinin (2017, s. 62) mukaan yksittäistä ohjelmakooditiedostoa tai joissain ohjelmointikielissä tietoa ja funktioita kerääviä kokonaisuuksia. Lisäksi Martinin (2017, s. 62) mukaan moduuleille tulisi päteä mm. yhden vastuun periaate (*Single Responsibility Principle*), joka on historiallisesti määritelty, että "moduulilla tulee olla yksi, ja vain yksi, syy muuttua".

Komponentit muodostavat isompia kokonaisuuksia, jotka Martinin (2017, s. 96) mukaan käännetään usein kirjastoiksi, kuten .NET-kielissä DLL-tiedostoiksi ja Javassa JAR-tiedostoiksi. Lisäksi komponentit pitäisi pystyä kehittämään ja viemään tuotantoon itsenäisesti. Martinin (2017, s. 67) mukaan yhden vastuun periaate pätee myös komponenteille, mutta komponenttitasolla tätä nimitetään yleisen sulkeuman periaatteeksi (*Common Closure Principle*). Martin (2017, s. 153) toteaa, että joissain tilanteissa komponentit täytyy erottaa palveluiksi kuten mikropalveluiksi.

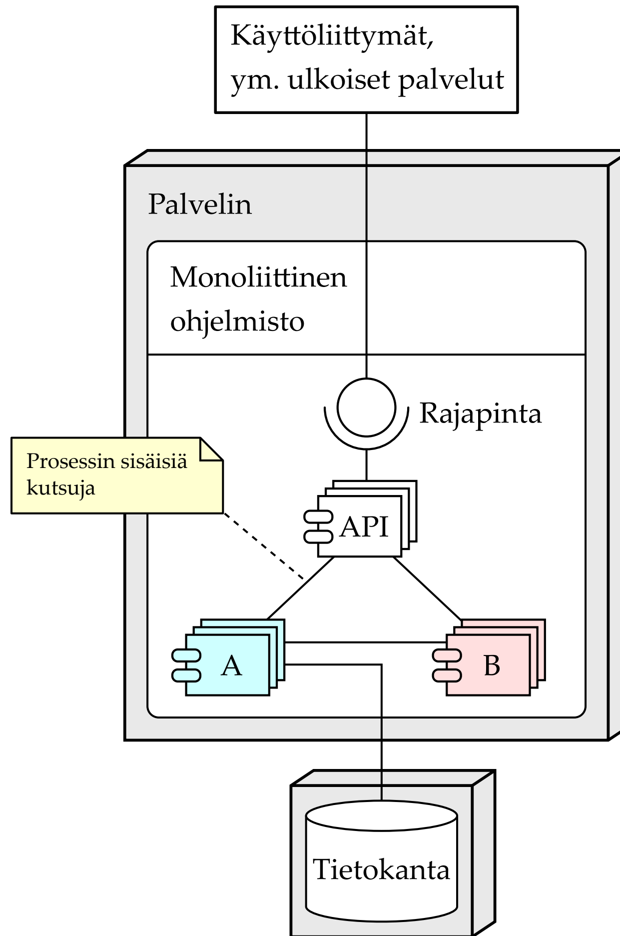
2.2 Monoliittiset tietokantapohjaiset rajapintapalvelut

Monoliittisille tietokantapohjaisille rajapintapalveluille on yhteistä tietokannan ja rajapinnan yhteys ohjelmistoon. Newman (2015, s. 41) mainitsee, että yleisin integrointitapa monoliittisille ohjelmistoille on ollut yksi jaettu tietokanta. Tyypillinen monoliittinen arkkitehtuuri voi Villamizarin ym. (2015) mukaan käyttää esimerkiksi yleisiä MVC-ohjelmistokehyksiä kuten JEE, .NET, Symfony, Rails, Grails tai Play.

Lisäksi Villamizarin ym. (2015) mukaan monoliittisille ohjelmistoille voidaan luoda sisäinen arkkitehtuuri esimerkiksi käyttäen kerrosarkkitehtuuria jakamalla ohjelmisto presentaatiokerrokseen, liiketoimintakerrokseen ja relaatiotietokantaan yhdistävään persistenssikerrokseen. Toisaalta monoliittisille ohjelmistoille on useita arkkitehtuureja, joista kerrosarkkitehtuuri on vain yksi tapa.

Yhtenevää monoliittisille tietokantapohjaisille rajapintapalveluille sisäisestä arkkitehtuurista huolimatta on tietokanta ja rajapinta. Kuviosta 1 ilmenee kuinka rajapinta ja tietokanta sijoittuvat. A-komponenttien joukko toteuttaa tietokantaan yhdistävät toiminnot. API-komponenttien joukko muodostaa ohjelmiston rajapinnan esimerkiksi HTTP:n päällä toimivan REST- tai GraphQL-rajapinnan. Ohjelmistossa on yleensä muutakin toimintaa esimerkiksi sovelluslogiikkaa, jota kuvaa B-komponenttien joukko.

Komponenttien välillä olevat kutsut ovat prosessin sisäisiä kutsuja, esimerkiksi funktiokutsuja, jotka suoritetaan palvelimen sisäisesti. Yksinkertaisissa ohjelmistoissa kuviossa 1 esiintyvät komponentit voisivat yhdistyä, esimerkiksi ohjelmisto voisi olla pelkästään yksi funktio, joka toimii rajapintana ja kutsuu tietokantaa. Vaihtoehtoisesti jos ohjelmisto noudattaisi esimerkiksi kerrosarkkitehtuuria olisi komponenteilla tietty riippuvuusjärjestys.



Kuvio 1. Arkkitehtuurista riippumaton monoliittisen tietokantapohjaisen rajapintapalvelun yleiskuva. Kuvio ottaa mallia Lewisin ja Fowlerin (2014) mikropalveluarkkitehtuuria käsittelevistä kuvioista.

3 Mikropalveluarkkitehtuurin esittely

Mikropalveluarkkitehtuurin määritelmä on huomattavan lavea ja sillä tarkoitetaan ohjelmistoa, jossa ohjelmiston jokainen osa on oma mikropalvelunsa (Lewis ja Fowler 2014; Dragoni ym. 2017). Käsitteenä mikropalveluarkkitehtuuri ei määrittele esimerkiksi mikä on mikropalveluiden välinen integrointimenetelmä, miten ulkoinen rajapinta toteutetaan tai miten jakaminen mikropalveluihin pitäisi suorittaa. Tässä luvussa esiteltyä mikropalveluarkkitehtuurin määritelmää on rajattu käyttämällä Taibin, Lenarduzzin ja Pahlin (2018) kirjallisuuskartoituksessa löytyneitä mikropalveluarkkitehtuurista tunnistettuja malleja.

Tietokantapohjaista rajapintapalvelua ajatellen malleista on valikoitu rajapintayhdyskäytävämalli (*API-Gateway pattern*) ulkoisen rajapinnan toteutukseen, palvelinpäänlöytömalli (*Server-side Discovery pattern*) mikropalvelujen väliseen yhdistämiseen, tietokanta-per-palvelu -malli (*Database-per-service pattern*) tietokantajakoihin. Mikropalvelujaot esitetään Newmannin (2015, s. 31) käyttämän rajatun kontekstin (*Bounded Context*) avulla. Se on osa Evansin kirjassa *Domain-Driven Design: Tackling Complexity in the Heart of Software* (2003) esiteltyä suunnittelumenetelmää, joka edeltää mikropalveluja.

3.1 Modulaarisuus jakamalla mikropalveluihin

Lewisin ja Fowlerin (2014) mukaan mikropalveluarkkitehtuurissa modulaarisuus muodostuu jakamalla ohjelmisto mikropalveluihin, joiden tulisi olla päivitettäviä ja korvattavia. Mikropalvelun tulisi Newmannin (2015, s. 1) mukaan noudattaa yhden vastuun periaatetta (*Single Responsibility Principle*), joka on esitelty myös monoliittisille ohjelmistoille moduuleille ja komponenteille luvussa 2.1. Tämän lisäksi hyviä mikropalvelun ominaisuuksia ovat Newmannin (2015, s. 30) mukaan löyhä kytkentä (*Loose Coupling*) ja korkea koheesio (*High Cohesion*).

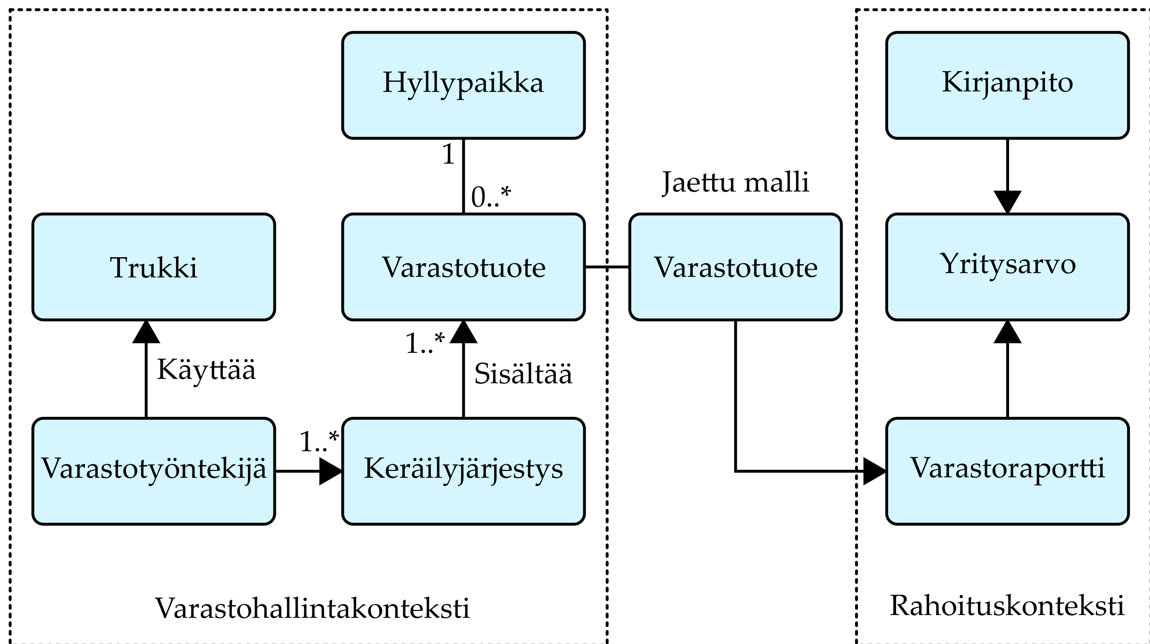
Löyhällä kytkennällä pyritään Newmannin (2015, s. 30) mukaan siihen, että mikropalvelulla olisi mahdollisimman vähän riippuvuuksia. Korkealla koheesiolla pyri-

tään Newmannin (2015, s. 30) mukaan siihen, että mikropalvelun toimintojen tulisi liittyä oleellisesti toisiin. Yhdessä näillä tavoitellaan, että mikropalvelu olisi mahdollisimman itsenäinen osa ohjelmistoa.

Mikropalveluiden välisten rajojen tunnistamiseen Newman (2015, s. 31) esittelee rajatun kontekstin käsitteen. Rajattu konteksti on alun perin esitelty osana Evansin (2003) sovelluslähtöistä suunnittelumenetelmää (*Domain-Driven Design*). Sittemmin myös Evans (2016) on todennut, että hänen esittelemänsä rajatut kontekstit soveltuvat mikropalvelujakojen hahmottamiseen. Huomattavaa on, että tämä suunnittelumenetelmä edeltää mikropalvelujen käsitettä, eikä se käsittele mikropalveluita vaan on kokonaisvaltaisempi menetelmä, jota voi soveltaa myös monoliittisiin ohjelmistoihin.

Rajattu konteksti koostuu sovellusalueen malleista (*Domain model*) jotka ovat Evansin (2003, s. 336) mukaan kullekin kontekstille omia. Newmannin (2015, s. 31) mukaan tämä voidaan käsittää niin, että kullakin kontekstilla on rajapintansa, joka päättää mitä malleja rajapinnasta näkyy ulospäin. Lisäksi Newmannin (2015, s. 34) mukaan rajattuja konteksteja pitäisi ajatella tiettyinä toimintojen kokonaisuuksina, eikä pelkästään tietovarastoina.

Newman (2015, s. 32) havainnollistaa rajattuja konteksteja seuraavalla esimerkillä, katso kuvio 2. Esimerkissä on kaksi rajattua kontekstia ja näiden välillä jaettu malli. Ensimmäisenä kontekstina on varastohallinnankonteksti, jonka mallit keräävät varastohallintaan liittyviä tietoja kuten hyllypaikan ja muodostaa varastotyöntekijälle tarvittavan keräysjärjestyksen. Toisena kontekstina on rahoituskonteksti, jonka mallit keräävät yrityksen rahoitukseen kuten kirjanpitoon ja yritysarvoon liittyviä tietoja.



Kuvio 2. Esimerkki mikropalvelujakoihin soveltuvista rajatuista konteksteista. Kuvio on suomennettu Newmannin (2015, s. 32 *Figure 3-1*) esittämästä kuviosta.

Keskeistä on, että nämä kontekstit eivät ole sidoksissa suoraan toisiinsa paitsi jaetun varastotuotemallin kautta. Jaetuissa malleissa tulisi Newmannin (2015, s. 32) mukaan olla vain tarvittavat tiedot, esimerkiksi rahoituskontekstissa varastotuotteesta ei tarvita hyllypaikan tietoa. Käytännössä varastohallintakontekstin rajapinta tuottaisi jaetun varastotuotemallin, jossa on vain tarpeelliset tiedot. Kun sovellusalueesta on tunnistettu rajatut kontekstit, niistä voidaan Newmannin (2015, s. 33) mukaan muodostaa mikropalvelut ohjelmistolle.

Mikropalvelun koko määräytyy usein jakolinjojen löytämisellä. Esimerkiksi jos jakolinjojen tunnistaminen ei onnistu voi mikropalvelu olla isompi kokonaisuus (Newman 2015, s. 33). Mikropalvelu voi olla myös hyvin pieni osa ohjelmistoa esimerkiksi liiketoimintaprosessi, joka kutsuu muita mikropalveluita. Jos mikropalvelu joutuu kommunikoimaan erityisen paljon toisen mikropalvelun kanssa, on mahdollista, että nämä tulisi yhdistää yhdeksi isommaksi mikropalveluksi (Newman 2015, s. 101).

3.2 Tietokanta kutakin mikropalvelua kohden

Tämä malli tunnetaan Taibin, Lenarduzzin ja Pahlin (2018) mukaan nimellä tietokanta-per-palvelu. Malli mainitaan myös useissa lähteissä (Lewis ja Fowler 2014; Newman 2015, s. 41-42; Dragoni ym. 2017), ja sen mukaan tietokantaa ei tulisi käyttää integrointimenetelmänä ja tietokannan tulee olla itsenäinen kullekin mikropalvelulle. Tästä syntyy selkeä ero monoliittisen ja mikropalveluarkkitehtuurin välillä, sillä monoliittisissa ohjelmistoissa jaettu tietokanta on yleinen tapa.

Toisaalta Taibi, Lenarduzzi ja Pahl (2018) mainitsevat, että jaettua tietokantaa voidaan käyttää mikropalveluissa, kun ollaan siirtymässä monoliittisesta ohjelmistosta mikropalveluarkkitehtuuriin. Siirryttäessä jaettu tietokanta voi helpottaa muutostyön tekemistä. Mikropalveluja sovellettaessa tietokantapohjaisiin ohjelmistoihin saattaa olla hyödyllistä ajatella mikropalvelujakoja tietokannan näkökulmasta.

3.3 Mikropalvelujen yhdistäminen ja integrointimenetelmät

Mikropalvelut tarvitsevat kommunikointitavan ja keinon, jolla ne yhdistetään toisiinsa. Taibin, Lenarduzzin ja Pahlin (2018) kirjallisuuskartoituksessa tunnistama palvelinpään löytämismalli (*Server-side Discovery*) tarjoaa keinon, jolla mikropalvelut löytävät ja yhdistyvät toisiinsa, mutta se ei ota kantaa esimerkiksi palvelujen väliseen kommunikointitapaan.

Palvelinpään löytämismalli koostuu palvelurekisteristä (*Service Registry*) ja kuormitustasaajasta. Kun uusi mikropalvelu käynnistetään, se rekisteröi itsensä palvelurekisterille löydettäväksi. Tämän jälkeen mikropalvelun kutsuessa toista mikropalvelua viestit kulkevat ensin kuormitustasaajalle, joka tarpeen vaatiessa kysyy palvelurekisteriltä mikropalvelun sijaintia. Sijainnin löydettyään kuormitustasaaja ohjaa viestin vastaanottavalle mikropalvelulle.

Yhdistämisen lisäksi tarvitaan mikropalvelujen välinen kommunikointitapa, jota kutsutaan yleisemmin integrointimenetelmäksi. Mikropalveluarkkitehtuuri ei määrää mitään tiettyä integrointimenetelmää mikropalvelujen välille vaan Lewisin

ja Fowlerin (2014) mukaan mikropalveluarkkitehtuurissa kommunikointiin tulisi käyttää kevyitä ratkaisuja kuten HTTP-resurssirajapintaa. Dragoni ym. (2017) mukaan integrointimenetelmät voidaan jakaa kahteen kategoriaan: orkestraatioisiin ja koreografisiin.

Orkestraatioisissa integrointimenetelmissä mikropalvelu viestii suoraan vastaanottavalle mikropalvelulle. Tällöin kutsujan ja kutsuttavan mikropalvelun välille syntyy kytkös, joka rikkoo luvussa 3.1 esitettyä mikropalveluiden löyhän kytkennän ominaisuutta. Tämän takia Dragonin ym. (2017) mukaan mikropalveluarkkitehtuurille on ominaista, että suositaan koreografisia integrointimenetelmiä, joissa kytkökset eivät ole vahvoja mikropalvelujen välillä.

Koreografisissa integrointimenetelmissä kommunikointi on tapahtumapohjaista. Mikropalvelut julkaisevat tapahtumia keskitetylle tapahtumajärjestelmälle, joka välittää viestin tapahtumaan liittyneille kuuntelijoille. Tässä tavassa ei synny kytkentää mikropalvelujen välille vaan viestin lähettäjä sekä kuuntelija on kytkettynä keskitettyyn viestintäohjelmistoon.

Viestintäohjelmistosta konkreettinen esimerkki voisi olla Bucchiaronen ym. (2018) kokemusreportaasissa käytetty RabbitMQ, myös Newman (2015, s. 55) mainitsee, että RabbitMQ soveltuu tähän. RabbitMQ perustuu viestijonoihin, joiden avulla viestit välitetään. Viestin lähettäjä julkaisee (*Publish*) nimettyyn viestijonoon haluamansa viestin. Viestin vastaanotto tapahtuu liittymällä kuuntelemaan (*Subscribe*) nimettyä viestijonoa. Jonoilla viestintäohjelmisto voi varmistaa, että viesti käsitellään vain kerran, vaikka mikropalvelusta olisi useampi ilmentymä kuuntelemassa samaa jonoa.

3.4 Ulkoisen rajapinnan toteutus

Taibin, Lenarduzzin ja Pahlin (2018) mukaan rajapintatoteutukselle suositeltu malli on rajapintayhdyskäytävä (*API-gateway*), joka tunnetaan myös toiselta nimeltään käyttöliittymän taustaohjelmistona (*Backend for Frontend*) (Newman 2015, s. 72). Mallin tärkeyttä korostaa myös se, että Taibin ja Lenarduzzin (2018) mukaan

rajapintayhdyskäytävän puute on tunnistettu yhdeksi mikropalvelujen pahoiksi hajuiksi.

Rajapintayhdyskäytävämallin mukaan mikropalveluja ei tulisi kutsua ohjelmiston ulkopuolelta suoraan, vaan ulkoisia rajapintoja varten luodaan rajapintayhdyskäytävä, johon ulkopuoliset kyselyt ohjataan. Rajapintayhdyskäytävä voi toteuttaa useamman rajapinnan eri asiakasohjelmia ajatellen, esimerkiksi omat rajapinnat mobiili- ja työpöytäsovelluksia varten. Lisäksi se pystyy muuttamaan mikropalvelujen kommunikointiprotokollan asiakasohjelmille sopivaksi.

Rajapintayhdyskäytävä voi toteuttaa rajapintakyselyille autentikointi tarkistuksia, monitorointia ja joissain tapauksissa se toimii kuormitustasaajana (*load balancer*) (Taibi, Lenarduzzi ja Pahl (2018)). Rajapintayhdyskäytävä ei kuitenkaan toimi mikropalvelujen hallitsijana tai muutenkaan välitä viestejä mikropalvelujen välillä vaan päätarkoitus on toteuttaa rajapinnat asiakasohjelmille.

3.5 Mikropalveluarkkitehtuurin vertailu monoliittiseen

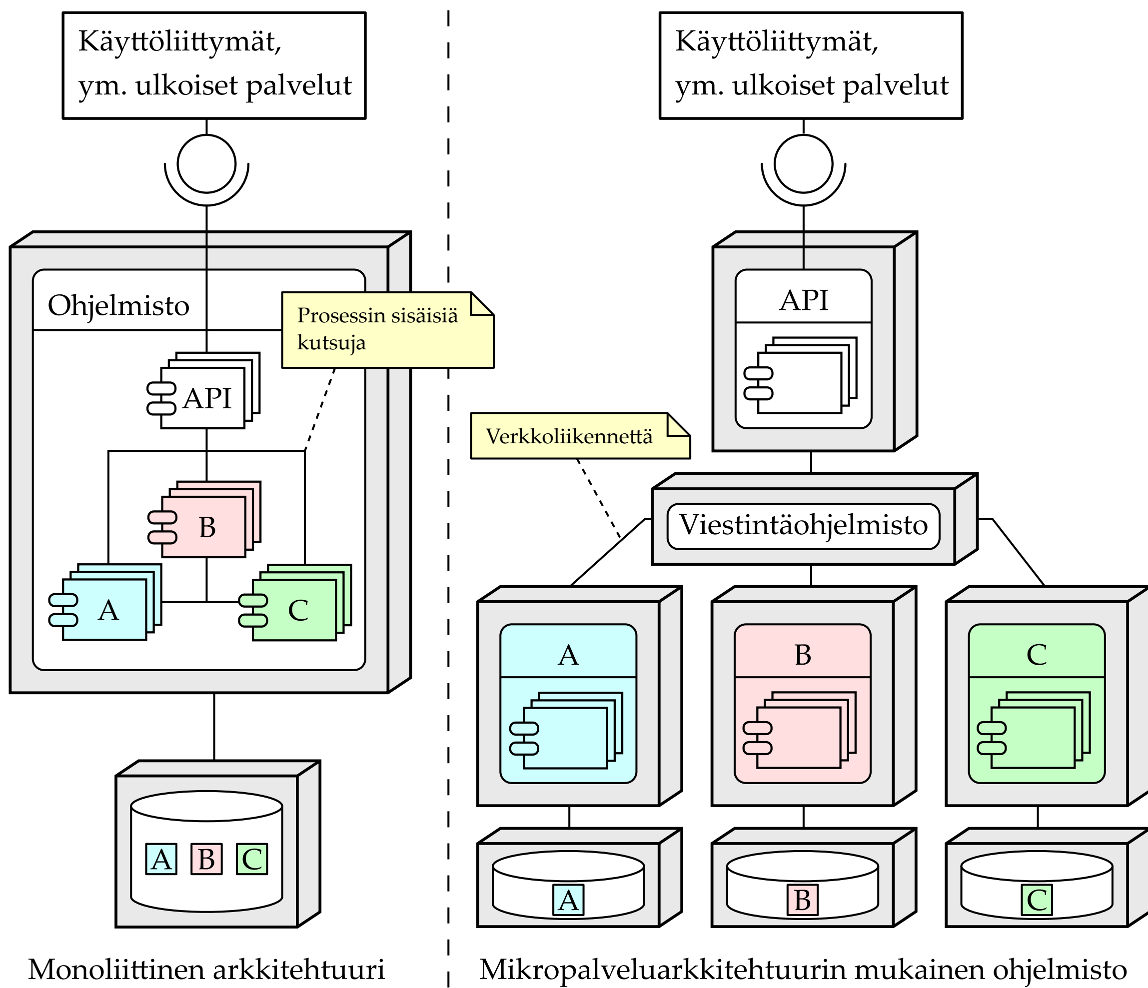
Mikropalveluarkkitehtuuri eroaa monoliittisesta usealla tavalla. Seuraavassa on esitelty pääeroja sijoittelukaavion avulla, katso kuvio 3. Kuviosta näkyy tuotantoon vietävät yksiköt harmaapohjaisina laatikoina, nämä voisi olla kontteja tai fyysisiä palvelimia. Kuvion vasemmalla puolella olevassa monoliittisessa ohjelmistossa on kaksi tuotantoon vietävää yksikköä, tämä vastaa luvussa 2.2 esiteltyä arkkitehtuuria. Oikealla puolella on mikropalveluarkkitehtuurilla vastaava ohjelmisto, jonka pääosat ovat viestintäohjelmisto sekä API-, A-, B- ja C-mikropalvelut, jotka on kaikki omia tuotantoon vietäviä yksiköitä.

Huomattavin ero liittyykin tuotantoon vietävien yksiköiden määrään, joita monoliittisessa ohjelmistossa on kaksi eli itse ohjelmisto ja tietokanta. Sen sijaan mikropalveluarkkitehtuurissa tuotantoon vietäviä yksiköitä on kukin tietokanta ja mikropalvelut, joita voi lukumäärällisesti muodostua Dragonin ym. (2017) mukaan jopa satoja, kun ohjelmisto jaetaan osiin.

Rajapinnan muodostaa luvussa 3.4 esitelty rajapintayhdyskäytävä, tämä näkyy kuviossa 3 mikropalveluna API. Mikropalveluarkkitehtuurissa rajapintayhdyskäytävä keskustelee viestintäohjelmistolle, joka välittää viestit. Sen sijaan monoliittisessa ohjelmistossa rajapinta on osa ohjelmistoa kuten luvussa 2.2 on esitelty.

Viestintäohjelmisto välittää tapahtumia ja kutsuu pyydettyjä mikropalveluja. Usein myös luvussa 3.3 esitelty palvelunpään löytämismalli täytyy olla toteutettuna jollain tasolla viestintäohjelmistossa tai sitä ympäröivässä verkossa. Monoliittisissa ohjelmistoissa viestintäohjelmistoa ei tarvita sillä kommunikointi tapahtuu prosessin sisäisinä funktiokutsuina.

Tietokannat näkyvät kuviossa 3 samoin kuin luvussa 3.2 esitelty tietokanta-per-palvelua malli. Huomattavin ero verrattuna monoliittiseen ohjelmistoon tässä on, että kullakin mikropalvelulla on oma tietokantansa. Esimerkiksi kuvion mikropalvelu A voisi olla luvun 3.1 mukaisesti toiminnallinen kokonaisuus kuten varastohallinta, jolla olisi oma tietokantansa. Monoliittisissa ohjelmistoissa tätä voisi vastata komponenttien joukko A, mutta ne olisivat kytketty jaettuun tietokantaan.



Kuvio 3. Monoliittisen ja mikropalveluarkkitehtuurin vertaileminen sijoittelukaavion avulla. Kuvio perustuu Lewisin ja Fowlerin (2014) esittämiin kuvioihin. Lisäksi kuvio kerää edeltävien lukujen oleellisia asioita samaan kuvioon.

4 Hyötyjen ja haittojen vertailu

Tässä luvussa käsitellään mikropalveluarkkitehtuurin hyödyt ja haitat verrattuna monoliittiseen toteutukseen. Hyödyt ja haitat käsitellään yleisellä tasolla ja osa on rajattu koskemaan tietokantapohjaisia rajapintapalveluita. Hyödyt ja haitat on jaettu kahteen alalukuun, teknologisiin ja ohjelmistotuotantoprosessiin kohdistuviin. Teknologisissa käsitellään hyötyjä ja haittoja ohjelmistosuunnittelun näkökulmasta ja tarkastellaan miten ne vaikuttavat ohjelmiston laatuominaisuuksiin. Ohjelmistotuotantoon liittyvät hyödyt ja haitat keräävät ohjelmistotuotantoprosessia ja organisaatiota koskevia hyötyjä ja haittoja.

4.1 Teknologiset hyödyt ja haitat

Vertailtaessa teknologisia hyötyjä ja haittoja pyritään ottamaan huomioon ohjelmiston laatuominaisuuksia kuten esimerkiksi suorituskyky, luotettavuus, saatavuus, turvallisuus, muunneltavuus, siirrettävyys, ylläpidettävyys ja uudelleenkäytettävyys. Laatuominaisuuksia voidaan käyttää Koskimiehen ja Mikkosen (2005, s. 221) mukaan arvioimaan arkkitehtuureja, ja ne soveltuvat täten hyöty- ja haittanäkökulman tarkasteluun.

4.1.1 Mikropalvelujakojen hankaluudet haittana

Ylläpidettävyys on eräs päämotivaattoreista Taibin ym. (2017) kyselyn mukaan siirryttäessä mikropalveluarkkitehtuurin. Jos mikropalvelujaot pystytään tekemään hyvin niin, mikropalvelut lisäävät ohjelmiston ylläpidettävyyttä, sillä ne voidaan testata ja ottaa käyttöön itsenäisesti. Lisäksi ohjelmakoodin ymmärrettävyys lisääntyy koska mikropalvelut pyrkivät olemaan mahdollisimman pieniä kokonaisuuksia (Taibi, Lenarduzzi ja Pahl 2017).

Hyvien mikropalvelujakolinjojen löytäminen on Taibin ja Lenarduzzin (2018) kyselytutkimuksen mukaan kuitenkin tunnistettu yhdeksi suurimmaksi haasteeksi, jota

mikropalveluarkkitehtuurissa voi kohdata. Newman (2015, s. 33) huomauttaakin, että jos ei ole selvää miten ohjelma tulisi jakaa osiinsa, se kannattaa pitää monoliittisena aluksi, sillä väärien jakolinjojen tekeminen saattaa koitua kalliiksi.

Lisäksi luvussa 3.1 esitelty sovelluslähtöinen suunnittelumenetelmä jakolinjojen löytämiseksi ei ole ongelmaton, Rademacherin, Sorgallan ja Sachwehin (2018) mukaan haasteita syntyy, kun mikropalvelujen välillä tarvitaan muutoksia. Lisäksi menetelmällä tunnistetut sovellusalueen mallit voivat olla puutteellisia rajapintojen ja protokollamäärittelyjen osalta, mikä voi hankaloittaa arkkitehtuurin määrittelyä (Rademacher, Sorgalla ja Sachweh 2018).

Verrattuna monoliittisiin ohjelmistoihin mikropalveluarkkitehtuurin modulaarisuus on hankalampi toteuttaa. Koska monoliittisissa ohjelmistoissa modulaarisuus toteutetaan ohjelmakooditasolla, niihin on käytettävissä useita valmiita työkaluja. Esimerkiksi staattisia analysointi- ja refaktorointityökaluja, mitkä helpottavat ohjelmiston muunneltavuutta.

4.1.2 Mikropalvelujen uudelleenkäyttö ja päivitettävyys hyötynä

Mikropalvelujen uudelleenkäytölle ja päivitettävyydelle tarjoavat pohjan luvussa 3.1 esitetyt löyhä kytkentä ja korkea koheesio. Nämä helpottavat uudelleenkäyttöä ja päivitettävyyttä koska riittää, että käyttävä ohjelmisto osaa kommunikoida mikropalvelun rajapinnan kanssa. Tätä tukee myös Gouigouxin ja Tamzalitin (2017) kokemusraportti, jonka perustella uudelleenkäyttö kasvoi huomattavasti siirryttäessä monoliittisesta arkkitehtuurista mikropalveluarkkitehtuurin, sillä joitain mikropalveluja pystyttiin uudelleenkäyttämään uusissa ohjelmistoissa.

Mikropalvelujen itsenäisyys helpottaa myös niiden korvaamista, Gouigouxin ja Tamzalitin (2017) kokemusraportissa havaittiin, että hyvin muodostetut mikropalvelut olivat huomattavasti helpompi korvata uudella versiolla kuin monoliittisessa ohjelmistossa, muutosten yhteydessä tehtävä laatuvarmistaminen nopeutui muutamista viikoista päiviin. Tämä johtui siitä, että koko ohjelmistoa ei jouduta laatuvarmistamaan käytettäessä mikropalveluita.

Tizzein ym. (2017) mukaan monen asiakkaan (*multi-tenant*) ohjelmistossa tuoterunkoarkkitehtuuria (*Product-line engineering*) ja mikropalveluarkkitehtuuria yhdistelemällä voidaan saavuttaa uudelleenkäyttöä myös ohjelmakooditasolla. He mainitsevat, että 62% ohjelmakoodista pystyttiin uudelleenkäyttämään uudessa arkkitehtuurissa. Toisaalta annettu prosentuaalinen luku on hankala vertailukohde, sillä tutkimus ei sisällä vastaavaa arvoa monoliittisesta ohjelmistosta. Lisäksi he summaavat tutkimuksessaan, että siirtyminen monoliittisesta ohjelmistosta mikropalveluarkkitehtuuriin vähenti ylläpitotarvetta ja mahdollisti skaalaamisen.

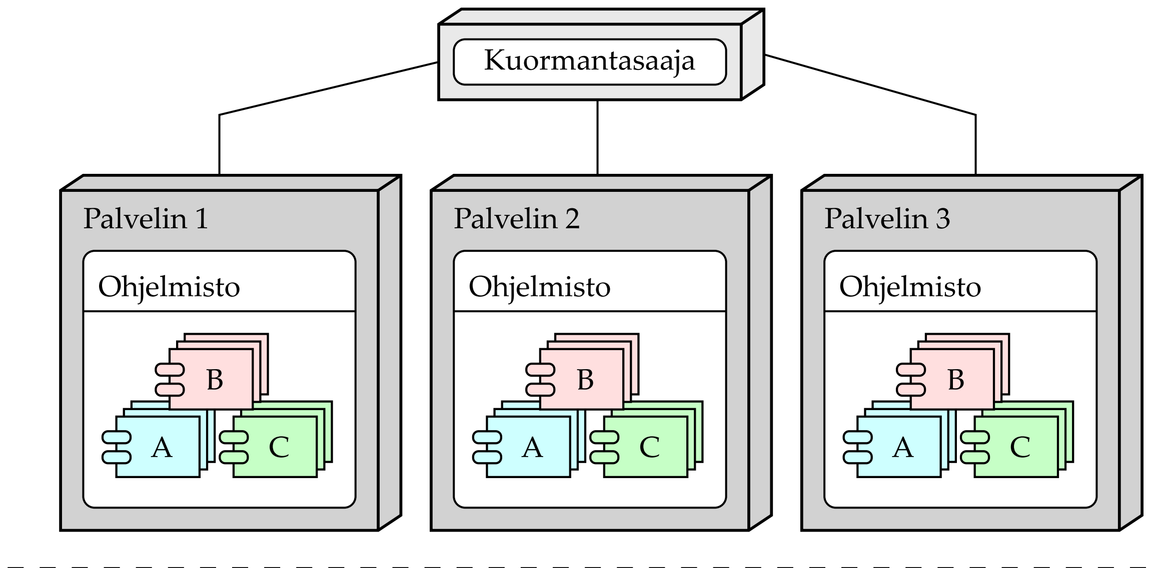
Monoliittisissa ohjelmistoissa Dragonin ym. (2017) mukaan muutokset moduuleissa tai komponenteissa vaativat ohjelman kääntämisen tai käynnistämisen uudelleen, toisin kuin mikropalveluarkkitehtuurissa. Samoin Dragonin ym. (2017) mukaan monoliittisissa ohjelmistoissa komponenttien päivittäminen voi aiheuttaa niin kutsutun “riippuvuus helvetin” jossa päivitykset voivat aiheuttaa yhteensopimattomuutta ja järjestelmä ei välttämättä edes voida kääntää ilman lisätyötä.

4.1.3 Skaalattavuuden hyödyt ja saatavuuden parantuminen

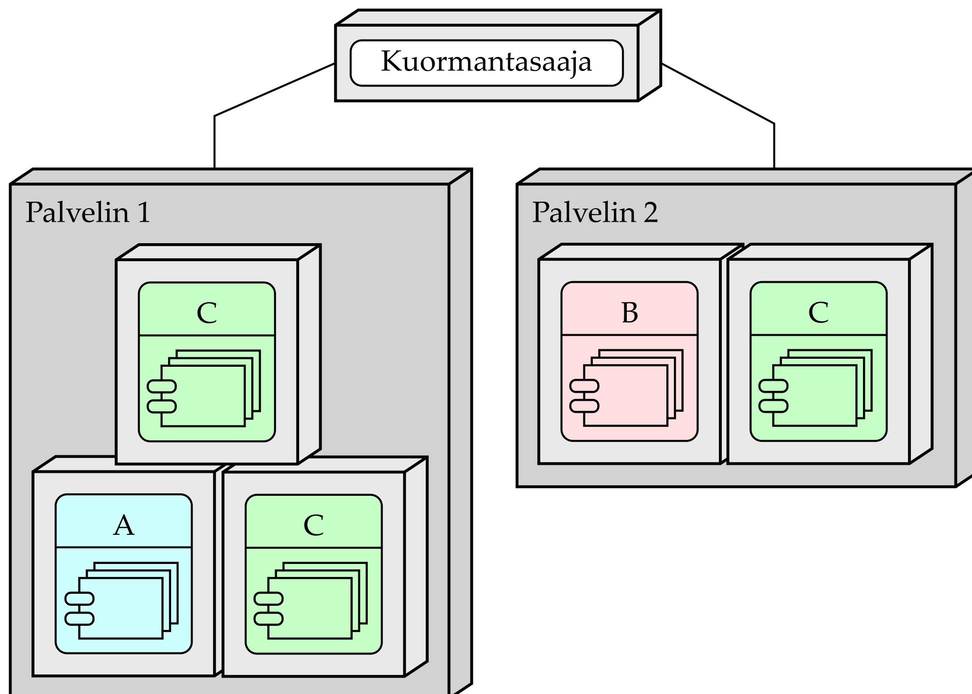
Skaalattavuus on eräs päämotivaattori Taibin, Lenarduzzin ja Pahlin (2017) kyselytutkimuksen mukaan siirryttäessä monoliittisesta arkkitehtuurista mikropalveluarkkitehtuuriin. Mikropalveluarkkitehtuurissa skaalattavuus muodostuu Dragonin ym. (2018) mukaan yksittäisten mikropalvelujen skaalattavuudesta. Kutakin mikropalvelua voidaan skaalata lisäämällä mikropalveluja samalle palvelimelle tai jakamalla niitä uusille palvelimille. Monoliittisissa ohjelmistoissa skaalattavuus kärsii koska koko ohjelmisto joudutaan skaalaamaan yhtenä kokonaisuutena.

Kuvio 4 perustuu Dragonin ym. (2018) skaalaamista tutkivan artikkelin kuvioon. Kuviosta ilmenee kuinka C-joukon komponenttien skaalaamistarve aiheuttaa monoliittisessa ohjelmistossa tarpeen skaalata koko ohjelmistoa. Sen sijaan mikropalveluarkkitehtuurissa voidaan C-mikropalvelua skaalata vain tarpeen mukaan, jolloin palvelimelle tuleva kuorma voi olla pienempi. Lisäksi A- ja B-mikropalvelusta ei tarvitse käynnistää enempää ilmentymiä kuten monoliittisessa ohjelmistossa.

Skaalautuminen monoliittisissä ohjelmistoissa



Skaalautuminen mikropalveluarkkitehtuurin mukaisissa ohjelmistoissa



Kuvio 4. Esimerkki monoliittisen ja mikropalveluarkkitehtuurin skaalautuvuudesta komponentin ja mikropalvelun C suhteen. Kuvio on suomennettu Dragonin ym. (2018, s. 98 *Fig. 1.*) artikkelista.

Skaalaaminen parantaa ohjelmiston saatavuutta Dragonin ym. (2018) mukaan koska mikropalvelut voidaan jakaa eri palvelinkeskuksiin ja kuorma voidaan tasata ohjelmiston eri osien kesken paremmin. Lisäksi Dragonin ym. (2018) mukaan saatavuus paranee myös ohjelmistopäivitysten yhteydessä, koska mikropalveluarkkitehtuurilla toteutettua ohjelmistoa ei tarvitse käynnistää kokonaan uudelleen päivitysten ajaksi kuten monoliittisissa ohjelmistoissa.

4.1.4 Tietoturvahaasteet haittana

Dragonin ym. (2017) mukaan tietoturva on ongelmallista hajautetuissa järjestelmissä kuten mikropalveluissa ja palvelusuuntautuneissa arkkitehtuureissa. Verrattuna monoliittisiin ohjelmistoihin, joissa komponenttien väliset kutsut ovat ohjelmiston sisäisiä funktiokutsuja, mikropalveluarkkitehtuurissa jokainen mikropalvelujen välinen kutsu täytyy kulkea verkon lävitse. Tästä syystä mikropalvelujen väliset yhteydet pitää turvata verkkotasolla, ja poikkeustilanteet täytyy hallita hajautetusti.

Yarginan ja Baggen (2018) mukaan mikropalveluarkkitehtuurien tietoturvahaasteet ja -uhat voidaan kategorisoida seuraavasti: laitteistoon-, virtualisaatioon-, käytettyyn pilvipalveluun-, kommunikaatioon-, palvelukerrokseen- ja orkestraatioon liittyvät uhat. Näistä kategorioista erityisesti kommunikaatioon ja orkestraatioon liittyviä uhkia ei esiinny monoliittisissa ohjelmistoissa.

Kommunikaatiouhiksi lukeutuu Yarginan ja Baggen (2018) mukaan esimerkiksi mikropalvelujen väliseen integraatiomenetelmään liittyvät uhat kuten salakuuntelu, sessiokaappaukset, väärennetty identiteetti, palvelunestohyökkäykset, väliintulohyökkäykset sekä muita verkkotason hyökkäyksiä kuten TLS-salaukseen liittyvät hyökkäykset. Monoliittisissa ohjelmistoissa komponenttien välinen kommunikointi tapahtuu prosessin sisäisinä kutsuina kuten funktiokutsuina ja ei kärsi komponenttitasolla näistä uhista.

Orkestraatioon kuuluu useamman mikropalvelun hallinta, koordinaatio ja automatisointi joihin vaikuttavia uhkia ovat Yarginan ja Baggen (2018) mukaan esimerkiksi löytämismalliin ja siihen liittyviin osiin kuten palvelurekisteriin kohdistuvat uhat.

Hyökkäykset saatavat lisätä tuntemattomia palveluja verkkoon tai ohjata liikennettä odottamattomilla tavoilla. Monoliittisissa ohjelmistoissa komponentit yhdistetään esimerkiksi ohjelmakooditasolla olevilla rajapinnoilla, joten näitä uhkia ei esiinny monoliittisissa ohjelmistoissa.

Tietoturvan ongelmallisuutta tukee myös Soldanin ym. (2018) tekemä kirjallisuuskatsaus, jossa tietoturva nousi yhdeksi suunnittelun kipukohdaksi verrattaessa monoliittisiin ohjelmistoihin. Katsauksesta ilmeni, että ongelmia aiheuttaa erityisesti mikropalvelujen välinen liikenne, joka tulee turvata verkkotasolla, ja mikropalveluista syntyvä rajapintojen suuri määrä, joka aiheuttaa ongelmia tietoturvan suunnittelemiselle. Kokonaisuutena mikropalveluarkkitehtuurissa kohdataan monia tietoturvahaasteita, jotka eivät koske monoliittisia ohjelmistoja, joten tämä on selkeä haitta, vaikka onkin hallittavissa.

4.1.5 Tietokannan hajautumisen haitat

Soldanin ym. (2018) kirjallisuuskatsauksessa ilmeni, että kehityksen aikaisesti ongelmia aiheuttaa eniten tietokannan hajautuminen ohjelmiston sisällä. Tietokanta hajautuu koska mikropalveluarkkitehtuurille on tyypillistä, että kullakin mikropalvelulla on oma tietokanta, tämä malli on esitelty luvussa 3.2. Hajautumisessa ongelmallisinta on Soldanin ym. (2018) mukaan tiedon eheyden säilyttäminen, useamman mikropalvelun välillä vaadittavat transaktiot ja tietyt hankalammat tietokantakyselyt.

Tiedon eheyttä tarkastelleet Furda ym. (2018) totesivat, että ongelmia tulee muutettaessa monoliittisessa ohjelmistossa olevia peräkkäisiä tietokantaa käsitteleviä käskyjä mikropalveluille sopiviksi. Tämä johtuu siitä, että monoliittisissa ohjelmistoissa peräkkäisten käskyjen välillä voidaan varmistaa, että tiedon eheys säilyy. Sen sijaan mikropalveluarkkitehtuurissa jokaisen käskyn välissä tai sen aikana saattaa eri ilmentymien välillä tapahtua muutoksia, joka rikkoo eheyttä.

Tietokantatransaktiot ovat keino, jolla eheyttä on voitu hallita. Näiden avulla käskyt voidaan jakaa atomiseen osaan, joka varmistaa, että kysely tai muutos tehdään

yhdellä kertaa. Mikropalveluarkkitehtuurissa transaktiot ovat ongelmallisia koska tietokanta hajautuu. Newman (2015, s. 91) ehdottaakin kompensoimaan tietokantatransaktioita, esimerkiksi tekemällä taustaprosesseja, jotka tarkistavat ja korjaavat eheyden. Yleisemmin tätä kutsutaan lopulliseksi eheydeksi (*Eventual Consistency*), jossa jonkin muun toiminnon tuloksena tietokanta lopulta eheytetään.

Tietokannan hajautumisen ongelmat on myös tiedossa yleisesti hajautetuissa järjestelmissä. Newman (2015, s. 232) viittaa CAP-teoreemaan, jonka mukaan on mahdotonta, että hajautettu järjestelmä voi samanaikaisesti olla eheä (*Consistency*), saatava (*Availability*) ja ositukselle sietokykyinen (*Partition-tolerance*), vain näistä kaksi on mahdollista saavuttaa hajautetuissa järjestelmissä samanaikaisesti. Newman (2015, s. 235) ehdottaa harkitsemaan tilannekohtaisesti mitkä näistä on tavoiteltavia ominaisuuksia kullekin mikropalvelulle.

4.2 Ohjelmistotuotannolliset hyödyt ja haitat

Ohjelmistotuotanto sisältää Haikalan ja Mikkosen (2011, s. 12) mukaan tietokoneohjelmiston tuottamiseen tarvittavia ohjelmistotyövaiheita ja tuotantoprosesseja. Tarkoituksena on tuottaa ohjelmisto, joka vastaa asiakkaan kohtuullisia odotuksia kehityskustannukset ja aikataulu huomioon ottaen (Haikala ja Mikkonen 2011, s. 12). Tässä luvussa käsitellään mikropalveluarkkitehtuurin hyötyjä ja haittoja ohjelmistotuotannollisesta näkökulmasta verrattuna monoliittiseen ohjelmistoon.

4.2.1 Automaation lisääntyminen ja aloituskustannusten nousu

Mikropalveluarkkitehtuurin käyttöönotto on Newmannin (2015, s. 103) mukaan huomattavan hankalaa verrattuna monoliittisen ohjelmistoon. Tämä johtuu siitä, että mikropalveluarkkitehtuuri vaatii automaatiota, jotta hajautetun ohjelmiston kompleksisuutta voidaan hallita. Balalaien, Heydarnoorin ja Jamshidin (2016) mukaan automatisoitu jatkuva integrointi ja toimitus ovat lähes vaatimuksena mikropalveluarkkitehtuurin käytölle sillä jokainen mikropalvelu voidaan viedä tuotantoon itsenäisesti.

Jatkuvalla integroinnilla (*Continuous integration*) tarkoitetaan Haikalan ja Mikkosen (2011, s. 175) mukaan, että kehityksessä olevat muutokset viedään jatkuvasti versionhallintaan ja testataan automaattisesti. Jatkuvalla toimituksella (*Continuous delivery*) tarkoitetaan, että ohjelmisto julkaistaan tuotantoon mahdollisimman automatisoidusti haluttuna hetkenä. Yhdessä jatkuva integraatio ja toimitus kuuluvat DevOps-käytäntöihin (Balalaie, Heydarnoori ja Jamshidi 2016).

DevOps-käytäntöjä ja tuotantoon viennin automatisointia voidaan hyödyntää myös monoliittisissa ohjelmistoissa, mutta automatisointi on tällöin yksinkertaisempaa, sillä tuotantoon vietäviä yksiköitä on käytännössä ohjelmisto ja tietokanta. Mikropalveluarkkitehtuurissa tuotantoon vietäviä yksiköitä on usein kymmenittäin kuten luvussa 3.5 todetaan. Automatisoinnin tuloksena myös tuotantoon vienti nopeutuu verrattuna monoliittisiin ohjelmistoihin, sillä kukin mikropalvelu voidaan viedä tuotantoon itsenäisesti.

Kalske, Mäkitalo ja Mikkonen (2018) ovat tutkineet haasteita siirryttäessä mikropalveluarkkitehtuuriin. He summaavat tutkimuksessaan, että siirtyminen vaatii huomattavia ponnistuksia organisaatiollisesti ja teknologisesti, lisäksi he kehottavat arvioimaan kuluja ja siirtymiseen liittyviä hyötyjä. Onkin oletettavaa, että mikropalveluarkkitehtuurin haasteet kuten vaadittava automatisointi lisäävät kuluja, kunnes kehittäjät ja organisaatio ovat tottuneet mikropalveluarkkitehtuuriin.

Kulujen nousua tukee Taibin ym. (2017) tekemä kysely, jonka mukaan mikropalveluarkkitehtuurilla kehitetty ohjelmisto vaatii enemmän aloituskustannuksia kuin monoliittinen ohjelmisto. Kyselyyn osallistujista 24% ilmoitti prosessien alkuun saattamiseksi vaativien kulujen nousevan 0-10% ja 76% ilmoitti kulujen nousevan 20-30%. Toisaalta kyselyssä nousi myös esille, että mikropalveluarkkitehtuurista koituvilla hyödyillä saatiin aloituskustannukset takaisin prosessien helpottuessa. Kyselyyn osallistujista 33% ilmoitti saavansa alkupanostukseen vaaditut kulut takaisin kahdessa vuodessa ja 66% kolmessa vuodessa.

4.2.2 Organisaatiorakenteen skaalautuvuus hyötynä

Lewisin ja Fowlerin (2014) mukaan mikropalveluihin jakaminen ja organisaatiorakenne seuraavat toisiaan siten, että kukin mikropalvelu täyttää tietyn liiketoiminnallisen tarpeen, jolla on muista riippumaton tiimi. Tämä on myös yksi syy, jonka takia Taibin ym. (2017) kyselyn mukaan mikropalveluarkkitehtuuriin siirrytään, sillä tällä pyritään lisäämään tiimien vastuuta omasta työstä. Vastuunjako mikropalvelujen kesken mahdollistaa suurien tiimien jakamisen pienempiin ja tehokkaampiin tiimeihin (Taibi, Lenarduzzi ja Pahl 2017). Newmannin (2015, s. 192) mukaan esimerkiksi Netflix ja Amazon ovat kehittäneet organisaatiorakennetta tähän tapaan, jotta vastuu tuotantoon vietävistä yksiköistä jakaantuu kullekin tiimille.

Kuviosta 5 ilmenee kuinka monoliittinen ja mikropalveluarkkitehtuuri eroavat organisaation osalta. Monoliittisissa ohjelmistoissa organisaatiorakenne muodostuu tuotantoon viennin ympärille ja tiimit ovat Kalsken, Mäkitalon ja Mikkosen (2018) mukaan usein keskittyneitä tiettyyn työalueisiin kuten tietokantoihin, käyttöliittymiin, palvelinohjelmistoihin, tuotehallintaan, laadunvarmistamiseen ja testaamiseen tai operaatiollisiin toimintoihin.

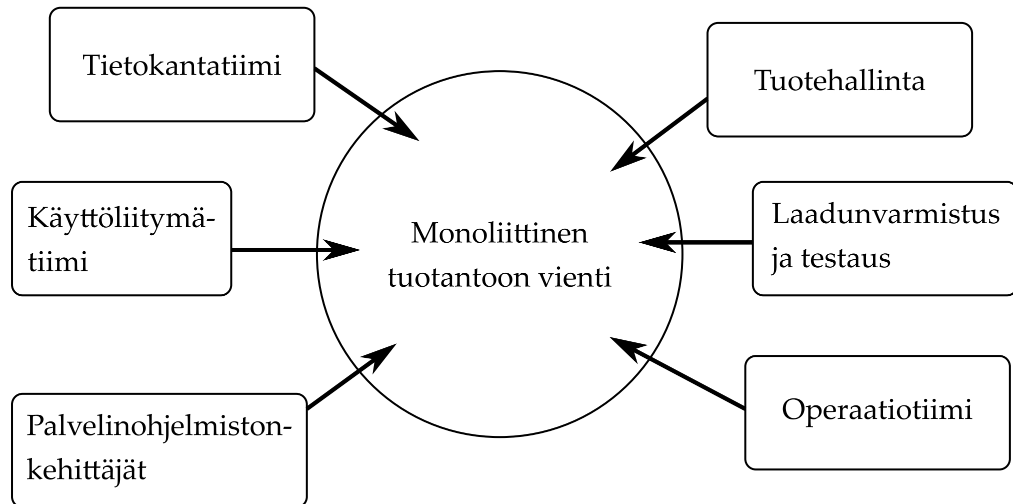
Mikropalveluarkkitehtuurissa organisaatio rakentuu mikropalvelukohtaisesti, esimerkiksi laskutusmikropalvelu muodostaisi oman laskutustiimin. Suurin ero tässä on, että kutakin työaluetta kohden ei useinkaan ole omaa työntekijää, vaan kukin mikropalvelukeskeinen tiimin pitää pystyä hallitsemaan useampaa osaamisaluetta, jotka vastaavat työalueita monoliittisessa ohjelmistossa. Vaikka kuviossa 5 on sisällytetty käyttöliittymäosaaminen kuhunkin mikropalveluun, on tämä hieman avoin ongelma sillä kirjallisuus ei käsittele käyttöliittymän jakamista mikropalvelujen mukaisesti.

Henkilömäärät tiimiä kohden ovat viitteellisiä. Newman (2015, s. 192) mainitsee Amazonin kahden pitsan säännön, minkä mukaan kukin tiimi pitää pystyä ruokkimaan kahdella pitsalla. Käytännössä tämä tarkoittaa noin kahdesta neljään henkilöä olevia tiimejä, mikä näkyy kuvion 5 mikropalveluissa. Singletonin (2016) mukaan vasta organisaatiot, joissa on vähintään 60 henkeä hyötyvät mikropalveluarkkiteh-

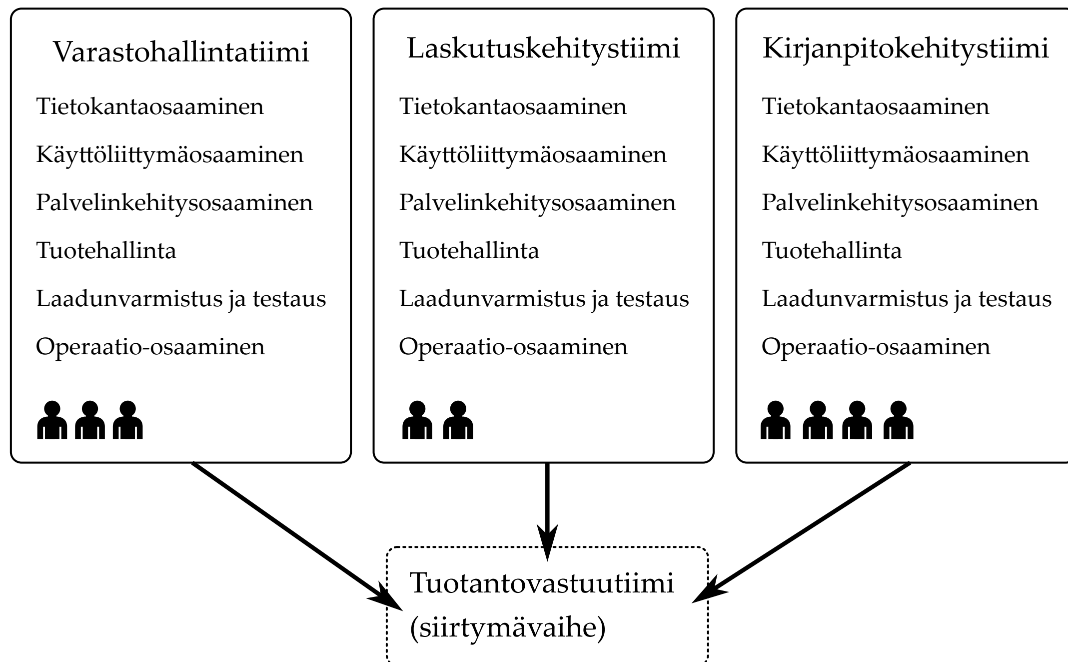
tuurista, mikä on hieman ristiriitaista sen kanssa, että Taibin ym. (2017) mukaan myös pienet yritykset harkitsevat mikropalveluarkkitehtuurien käyttöä.

Newman (2015, s. 202) ehdottaa pitämään siirtymävaiheessa oman tuotantovastuutiimiin, jotta siirtymä monoliittisesta ohjelmistosta ei olisi liian iso askel organisaatiolle. Tarkoituksena olisi kuitenkin päästä eroon erillisestä tuotantovastuutiimistä, sillä tuotantovastuu tulisi olla jokaisella mikropalvelukohtaisella tiimillä itsellään. Organisaatiollisesta näkökulmasta päähyödyt mikropalveluarkkitehtuurista on vastuun jakautuminen tiimeille ja organisaation skaalautuminen liiketoiminnallisten tarpeiden kasvaessa.

Organisaatiorakenne monoliittisissä ohjelmistoissa



Organisaatiorakenne mikropalveluarkkitehtuurin mukaisissa ohjelmistoissa



Kuvio 5. Monoliittisen ja mikropalveluarkkitehtuurin organisaatiorakenteet. Kuvio perustuu Kalsken, Mäkitalon ja Mikkosen (2018, s. 43 *Fig. 5.*, s. 44 *Fig. 6.*) esittämiin kuvioihin. Lisäksi kuviossa on Newmannin (2015, s. 202) mainitsema väliaikainen tuotantovastuutiimi.

5 Yhteenveto

Tämä tutkielma tarkasteli mikropalveluarkkitehtuurin ja monoliittisen ohjelmiston eroja, sekä pyrki vastaamaan asetettuun tutkimuskysymykseen mikropalveluarkkitehtuurin hyödyistä ja haitoista verrattuna monoliittisiin ohjelmistoihin. Käytettynä tutkimusmenetelmänä oli kirjallisuuskartoitus, jolla pyrittiin muodostamaan pohja kerätyille hyödyille ja haitoille. Tämän tutkielman rajoitteet liittyvät käsiteltyyn aineistoon, eikä voi olla täysin varma onko kaikki oleelliset hyödyt ja haitat löydetty.

Hyötyjen ja haittojen vertailun tuloksena näyttää, että mikropalveluarkkitehtuuri soveltuu tietokantapohjaisiin rajapintapalveluihin, jos tietyt ehdot täyttyvät. Erityisesti täytyy pystyä hallitsemaan organisaatiollisia ja teknologisia haasteita, jotta mikropalveluarkkitehtuurin hyödyt voidaan saavuttaa. Minkä takia mikropalveluarkkitehtuuri ei välttämättä sovellu pienille organisaatioille, sillä erinäiset haasteet ja alkuun pääseminen voi olla esteenä. Onkin ehkä järkevää suositella mikropalveluarkkitehtuuria vasta isommille organisaatioille.

Monoliittinen ohjelmisto voi olla edelleen soveltuvin tapa aloittaa ohjelmiston toteuttaminen, tähän viittaa myös osa kirjallisuudesta. Myöskään ei ole syytä, miksei monoliittista ohjelmistoa voisi jakaa vastaavasti liiketoiminnallisiin kokonaisuuksiin kuten mikropalveluarkkitehtuurissa. Vaikka tällä menetelmällä ei voitaisi saavuttaa täysin itsenäisiä tiimejä, niin ainakin osa hyödyistä voitaisiin saavuttaa monoliittisissakin ohjelmistoissa.

Mikropalveluarkkitehtuuriin siirtymistä käsittelevä kirjallisuus vaikuttaa olevan hyödyllinen lähde harkitessa mikropalveluarkkitehtuuria. Tässä tutkielmassa läpikäydyt lähteet ovat raportoineet onnistumisen merkkejä siirryttäessä. Toisaalta kirjallisuudesta ei näytä löytyvän tutkimustietoa, jossa olisi kuvattu epäonnistumisia siirtymävaiheessa. Siirtymävaiheiden epäonnistumiset sekä niiden pääsyyt voisikin olla eräs jatkotutkimuskohde, sillä useimmat eivät oletettavasti raportoi epäonnistumisia siinä määrin kuin onnistumisia.

Lähteet

Balalaie, Armin, Abbas Heydarnoori ja Pooyan Jamshidi. 2016. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". *IEEE Software* 33, numero 3 (toukokuu): 42–52. ISSN: 0740-7459. doi:10.1109/MS.2016.64.

Bucchiarone, A., N. Dragoni, S. Dustdar, S. T. Larsen ja M. Mazzara. 2018. "From Monolithic to Microservices: An Experience Report from the Banking Domain". *IEEE Software* 35, numero 3 (toukokuu): 50–55. ISSN: 0740-7459. doi:10.1109/MS.2018.2141026.

Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin ja Larisa Safina. 2017. "Microservices: Yesterday, Today, and Tomorrow". Teoksessa *Present and Ulterior Software Engineering,* toimittanut Manuel Mazzara ja Bertrand Meyer, 195–216. Cham: Springer International Publishing. ISBN: 978-3-319-67424-7. doi:10.1007/978-3-319-67425-4_12.

Dragoni, Nicola, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin ja Larisa Safina. 2018. "Microservices: How To Make Your Application Scale". Teoksessa *Perspectives of System Informatics,* toimittanut Alexander K. Petrenko ja Andrei Voronkov, 10742:95–104. Cham: Springer International Publishing. ISBN: 978-3-319-74312-7. doi:10.1007/978-3-319-74313-4_8.

Evans, Eric. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* 1. painos. Boston: Addison-Wesley Professional, 30. elokuuta. ISBN: 978-0-321-12521-7.

———. 2016. "DDD and Microservices: At Last, Some Boundaries!" 16. huhtikuuta. Viitattu 21. maaliskuuta 2019. https://www.infoq.com/presentations/ddd-microservices-2016.

Furda, A., C. Fidge, O. Zimmermann, W. Kelly ja A. Barros. 2018. "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency". *IEEE Software* 35, numero 3 (toukokuu): 63–72. ISSN: 0740-7459. doi:10.1109/MS.2017.440134612.

Gouigoux, J., ja D. Tamzalit. 2017. "From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture". Teoksessa *2017 IEEE International Conference on Software Architecture Workshops (ICSAW),* 62–65. Huhtikuu. doi:10.1109/ICSAW.2017.35.

Haikala, Ilkka, ja Tommi Mikkonen. 2011. *Ohjelmistotuotannon Käytännöt.* 12. uudistettu. Alma Talent. ISBN: 978-952-14-1754-2.

Jamshidi, P., C. Pahl, N. C. Mendonça, J. Lewis ja S. Tilkov. 2018. "Microservices: The Journey So Far and Challenges Ahead". *IEEE Software* 35, numero 3 (toukokuu): 24–35. ISSN: 0740-7459. doi:10.1109/MS.2018.2141039.

Kalske, Miika, Niko Mäkitalo ja Tommi Mikkonen. 2018. "Challenges When Moving from Monolith to Microservice Architecture". Teoksessa *Current Trends in Web Engineering,* toimittanut Irene Garrigós ja Manuel Wimmer, 32–47. Lecture Notes in Computer Science. Springer International Publishing. ISBN: 978-3-319-74433-9. doi:10.1007/978-3-319-74433-9_3.

Koskimies, Kai, ja Tommi Mikkonen. 2005. *Ohjelmistoarkkitehtuurit.* Talentum. ISBN: 952-14-0862-6.

Lewis, James, ja Martin Fowler. 2014. "Microservices: A Definition of This New Architectural Term". 25. maaliskuuta. Viitattu 29. tammikuuta 2019. https://martinfowler.com/articles/microservices.html.

Martin, Robert C. 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* 1 edition. London, England: Prentice Hall, 20. syyskuuta. ISBN: 978-0-13-449416-6.

Newman, Sam. 2015. *Building Microservices: Designing Fine-Grained Systems.* First Edition. Sebastopol, CA: O'Reilly Media. ISBN: 978-1-4919-5035-7.

Pautasso, C., O. Zimmermann, M. Amundsen, J. Lewis ja N. Josuttis. 2017. “Microservices in Practice, Part 1: Reality Check and Service Design”. *IEEE Software* 34, numero 1 (tammikuu): 91–98. ISSN: 0740-7459. doi:10.1109/MS.2017.24.

Rademacher, Florian, Jonas Sorgalla ja Sabine Sachweh. 2018. “Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective”. *IEEE Software* 35, numero 3 (toukokuu): 36–43. ISSN: 0740-7459. doi:10.1109/MS.2018.2141028.

Singleton, A. 2016. “The Economics of Microservices”. *IEEE Cloud Computing* 3, numero 5 (syyskuu): 16–20. ISSN: 2325-6095. doi:10.1109/MCC.2016.109.

Soldani, Jacopo, Damian Andrew Tamburri ja Willem-Jan Van Den Heuvel. 2018. “The Pains and Gains of Microservices: A Systematic Grey Literature Review”. *Journal of Systems and Software* 146 (1. joulukuuta): 215–232. ISSN: 0164-1212. doi:10.1016/j.jss.2018.09.082.

Taibi, Davide, ja Valentina Lenarduzzi. 2018. “On the Definition of Microservice Bad Smells”. *IEEE Software* 35, numero 3 (toukokuu): 56–62. ISSN: 0740-7459. doi:10.1109/MS.2018.2141031.

Taibi, Davide, Valentina Lenarduzzi ja Claus Pahl. 2017. “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation”. *IEEE Cloud Computing* 4, numero 5 (syyskuu): 22–32. ISSN: 2325-6095. doi:10.1109/MCC.2017.4250931.

———. 2018. “Architectural Patterns for Microservices: A Systematic Mapping Study”. Teoksessa *Proceedings of the 8th International Conference on Cloud Computing and Services Science,* 1:221–232. Maaliskuu. ISBN: 978-989-758-295-0. doi:10.5220/0006798302210232.

Tizzei, Leonardo P., Marcelo Nery, Vinícius C. V. B. Segura ja Renato F. G. Cerqueira. 2017. “Using Microservices and Software Product Line Engineering to Support Reuse of Evolving Multi-Tenant SaaS”. Teoksessa *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A on - SPLC ’17,* 205–214. Sevilla, Spain: ACM Press. ISBN: 978-1-4503-5221-5. doi:10.1145/3106195.3106224.

Villamizar, Mario, Oscar Garces, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas ja Santiago Gil. 2015. "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud". Teoksessa *2015 10th Computing Colombian Conference (10CCC),* 583–590. Bogota, Colombia: IEEE, syyskuu. ISBN: 978-1-4673-9464-2. doi:10.1109/ColumbianCC.2015.7333476.

Yarygina, T., ja A. H. Bagge. 2018. "Overcoming Security Challenges in Microservice Architectures". Teoksessa *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE),* 11–20. Maaliskuu. doi:10.1109/SOSE.2018.00011.

Zimmermann, Olaf. 2017. "Microservices Tenets". *Computer Science - Research and Development* 32, numero 3 (1. heinäkuuta): 301–310. ISSN: 1865-2042. doi:10.1007/s00450-016-0337-0.